Universidad
Carlos III de Madrid

Departamento de Ingeniería de Sistemas y Automática

TRABAJO FIN DE GRADO

CORRECCION DE ODOMETRÍA EMPLEANDO VISUAL SERVOING EN ROS

Alumno: *Juan José Gamboa Montero*

Tutor: *Pablo Marín Plaza*

Universidad Carlos III de Madrid

Índice

Lista de Figuras

Lista de Tablas

Agradecimientos

En primer lugar quiero agradecerle a Pablo, mi tutor, por todo el apoyo, la ayuda y la paciencia durante esta enorme y maravillosa aventura. También quería darle las gracias a Fernando, porque fue él me condujo hasta este proyecto y hasta el increíble grupo de investigadores que forman el LSI.

Quiero hacer una mención a todos aquellos que me han acompañado durante este viaje de 4 años, a todos los que han estado desde siempre ahí, a los que me han aguantado diariamente y también a los que han sabido estar cuando se les necesitaba aunque las agendas no coincidieran en la mayoría de los casos.

Por supuesto quiero dar las gracias a toda mi familia, que han sido una constante fuente de apoyo, ayuda y comprensión. Y especialmente quiero dar las gracias a mis padres, por dejarme hacer siempre lo que me ha gustado, educándome en la curiosidad y en la búsqueda de conocimiento; muchas gracias por haber confiado en mí durante todo este tiempo.

A todos, gracias de corazón.

Resumen

Los avances que se están realizando actualmente en el campo de los vehículos autónomos y la navegación no tripulada se hacen cada vez más patentes en nuestra sociedad. Desde las pequeñas incorporaciones que están haciendo las grandes marcas en sus modelos, como el aparcamiento asistido o la previsión de colisiones, hasta los pioneros en vehículos totalmente autónomos como son *Google* o *Uber*, la autonomía de los vehículos se perfila como una de los grandes avances de este tiempo. Algunos de los elementos que motivan este avance son la búsqueda del descenso de los accidentes en carretera, la reducción de la contaminación ambiental y el acceso a cualquier persona a un vehículo propio.

La Visión por Computador juega un papel fundamental en el avance de la autonomía en los vehículos, dotándolos de un gran flujo de información. Además, gracias a los últimos avances en sistemas de percepción, la fiabilidad de esta información se ha incrementado considerablemente.

Hasta ahora el acceso o la contribución en todo este tipo de tecnología estaba al alcance de muy pocos, pero con la aparición y creciente desarrollo de entornos de programación asociados a la robótica como ROS, basados en el código libre, estas barreras han desaparecido, permitiendo que se creen grandes comunidades de desarrolladores, que en definitiva contribuyen en una gran medida al ya vertiginoso avance de la robótica.

El proyecto propuesto consiste en crear un sistema que sirva para reiniciar el error que se genera en un sistema odométrico que controla el posicionamiento de un carrito de golf contribuyendo a su autonomía. El sistema se desarrollará en el entorno de programación conocido como ROS, empleando técnicas de Visión por Computador, específicamente de *Visual Servoing*, y siendo programado empleando el lenguaje de programación *C++*.

Palabras Clave: Vehículos Autónomos, Visión por Computador, ROS, código libre, Robótica, Odometría, *Visual Servoing*, Programación, *C++*.

Abstract

The progress that is currently being made in the field of autonomous vehicles and drone navigation are becoming more visible in our society. From small additions that are being introduced in some models of the big brands, such as the park-assist system or the collision forecast system, to the pioneers in completely autonomous vehicles like *Google* or *Uber*, vehicle autonomy is emerging as one of the major advances in this time. Some of the motivations for this advance are the pursuit of the drop in road accidents, reducing the pollution and the access to any individual to their own vehicle.

Computer vision plays a fundamental role in the progress of the autonomy in vehicles, providing them with a large flow of information. Besides, due to the latest developments in perception systems, the reliability of this information has increased considerably.

Until now, access or contribution to all this technology, was available to only a few people, but with the emergence and the development of programming environments associated to robotics such as ROS, based on open source code, all this obstacles have disappeared, allowing large communities of developers, which contribute to a great extent to the vertiginous advance of robotics.

The proposed project is to create a system that serves to reset the error that is generated in an odometer system that controls the positioning of an golf cart, contributing to its autonomy. The system will be developed in the programming environment known as ROS, using computer vision techniques, specifically Visual Sevoing, and being programmed using the programming language *C++*.

Keywords: Autonomous Vehicles, Computer Vision, ROS, Open-Source code, Robotics, Odometry, *Visual Servoing*, Programming, *C++*.

1. Introducción

De entre los últimos avances que se están realizando actualmente en el campo de la robótica, uno de los más llamativos y que poco a poco se está implantando más en la vida cotidiana es el de los vehículos autónomos. Los progresos tecnológicos en materia de procesamiento y reconocimiento de imágenes, además de en adquisición de datos del vehículo, están logrando que se avance poco a poco en este campo, situación que está propiciada por la constante necesidad de seguridad al volante. En la actualidad se están desarrollando y aplicando sistemas de asistencia al conductor a la hora de aparcar, tomar curvas o evitar obstáculos que puedan provocar daños a éste, peatones o a la vía pública. Se está buscando imitar e incluso superar la capacidad de reacción humana en los vehículos autónomos, siendo esto un factor determinante a la hora de prevenir accidentes. La conducción autónoma presenta múltiples ventajas, de las que destacan las siguientes [1]:

- **Tiempo libre**. La conducción autónoma ofrece al conductor la posibilidad de realizar otras tareas durante el trayecto, optimizando el mismo.

- **Aumento de la seguridad**. El error humano es un factor determinante en la mayor parte de las colisiones. Los fallos de atención del entorno, juzgar erróneamente el comportamiento o las reacciones de otros vehículos o un exceso de prisa en muchos casos son la causa de accidentes. Según el informe de la OMS de 2013 [2] aproximadamente 1.24 millones de personas mueren anualmente en la carretera y de 20 a 50 millones sufren daños no fatales.

- **Reducción de la emisión de gases**. Con los vehículos autónomos se busca un aprovechamiento del espacio de tráfico, reduciendo la congestión y ofreciendo tiempos de viaje más consistentes, a través del uso de tecnología de “vehículo conectado”.

- **Permite el acceso a vehículos a cualquier persona.** Hay mucha gente que no posee un carné de conducir o que no tiene acceso a un vehículo. Una persona discapacitada en algunos casos no puede conducir. Con el desarrollo de la autonomía en los vehículos se pretende incrementar la movilidad de estas personas.

Pese a estas ventajas que presenta la conducción autónoma, este tipo de vehículos aún se encuentra en desarrollo, ya sea porque en la mayoría de los casos se encuentran aún en fase de pruebas. Aún así, se trata actualmente de una opción de inversión muy atractiva y grandes marcas están paulatinamente dotando de mayor inteligencia a sus vehículos, o bien lanzando su propia gama de vehículos autónomos como es el caso de *Google* o *Uber* [3].

Ahora bien, un vehículo autónomo no es un concepto singular, o compuesto por un solo sistema, sino que se encuentra formado por un conjunto de sistemas que, intercomunicados, colaboran para que el vehículo realice la tarea del modo más eficaz posible, buscando reducir el margen de error al máximo. En el caso que se abordará en este proyecto el vehículo autónomo en cuestión posee un sistema odométrico para determinar la posición del mismo en un plano digital. La odometría lleva asociado un error, ya que se basa en el desplazamiento de

las ruedas del vehículo, de modo que cuanto más avance mayor será la incertidumbre asociada a su posición.

De este modo se pretende trabajar en esta faceta del conjunto, la de corregir y reiniciar el error que paulatinamente se produce a lo largo del desplazamiento de un vehículo autónomo, dando al vehículo un punto de apoyo sobre el que ir desarrollando el movimiento. Para ello, este apoyo deberá cumplir unos requisitos, entre ellos, que sea, como es lógico, una fuente fiable y por otro lado que sea económico y accesible. Los avances realizados en el campo de la percepción visual, aumentando la calidad de la detección, incluso en cámaras de baja resolución, hacen que trabajar dentro de este campo sea lo idóneo para la tarea que se anticipa.

Además de determinar el campo en el que se va a trabajar, que en este caso es el de la percepción visual, se necesita un entorno en el que se pueda desarrollar el programa en el que se basará el proyecto. Se necesita también que dicho entorno esté familiarizado con la versatilidad, permitiendo que en el trabajo que se desarrolle la portabilidad sea prioritaria, para que sea aprovechable para otros proyectos sin necesidad de hacer excesivas modificaciones en el código. Para ello se trabajará en el entorno de trabajo conocido como *ROS*, que reúne además de las cualidades buscadas una extensa variedad de bibliotecas, una comunidad en continuo crecimiento y siendo además de código libre [4], permitiendo que el coste de realizar la aplicación en términos de software sea prácticamente nulo.

El propósito del trabajo es diseñar un sistema que sea capaz de construir información acerca de la posición de un vehículo con respecto a un sistema de referencia global. Para ello se emplearán métodos de visión por computador con los cuales el sistema será capaz de construir una información fiable en tiempo real. Tras esto, la enviará como una referencia, corrigiendo el error que generan los métodos de posicionamiento principales del vehículo y contribuyendo a la autonomía del vehículo.

Para lograr este objetivo general se han de establecer una serie de metas a un menor nivel:

1. Implementar el entorno de trabajo virtual conocido como *ROS*, siendo para ello necesario la obtención de conocimientos básicos e intermedios acerca del mismo.

2. Desarrollar una aplicación que sea capaz de comunicarse tanto con otro programa encargado de la detección visual del entorno, como con un fichero de configuración donde se especifica la información absoluta de la posición y orientación de unas balizas. Estas balizas serán referencias que se irán colocando a lo largo del recorrido del vehículo.

3. A partir de estas dos fuentes de información geométrica, confeccionar un árbol de transformadas que permita conocer la posición global del vehículo a través de la información obtenida anteriormente.

4. Filtrar la información recibida para evitar que los falsos positivos pasen a través del ejecutable hacia el destino.

5. Enviar la información resultante con un cierto grado de confianza para que el ejecutable receptor sea capaz de gestionarla de una manera óptima.

2. Estado del Arte

Al haber establecido los principios básicos del proyecto, queda, por tanto, aislar el problema a solucionar, que en este caso está basado en emplear métodos de visión por computador para que, a través de una cámara web, se pueda determinar la posición y orientación exactas de dicha cámara con respecto a un sistema de referencia global. Para ello se ha de definir el concepto de visión por computador y abrir el abanico de técnicas surgidas a partir de este campo y justificar la selección de la que mejor se adapta al caso basándose de una serie de criterios.

Se define visión por computador como el campo perteneciente a la inteligencia artificial que emplea métodos para procesar, analizar y comprender imágenes e información multidimensional proveniente del mundo real para producir información simbólica o numérica, por ejemplo en forma de decisiones [5]. La motivación principal del desarrollo de este campo es la de replicar la capacidad de visión propia de una ser humano a través de la percepción y comprensión electrónica de una imagen [6]. Lo que se conoce como "comprensión" se podría definir como el descifrado de la información proveniente de una imagen empleando métodos multidisciplinares concernientes a campos como el de la geometría, física, estadística o aprendizaje automático [7]. De entre las múltiples aplicaciones en las que actualmente se emplea visión por computador destacan el control de procesos de producción, la detección de eventos, el modelado de entornos y objetos, sistemas de interacción humano-maquina o la navegación.

Pero, como se ha indicado a la hora de definir los objetivos principales del proyecto, no se trata de innovar o ahondar excesivamente dentro del campo de la visión por computador, sino de encontrar como aplicar esta disciplina al problema que se ha propuesto para lograr una solución que sea adecuada. Para ello hay que situarse dentro de una serie de condiciones, un marco, y desde ahí definir y seleccionar las opciones que se presentan. Esas opciones son las denominadas técnicas de visión por computador, y el marco en el que se sitúan es en el del entorno de trabajo, en este caso, *ROS*. *ROS* posee una gran biblioteca dedicada a la visión por computador, donde el software más representativo de cada una de estas técnicas ha sido adaptado para que se pueda trabajar de una manera libre y cómoda. A continuación se listan cada una de dichas técnicas y el software con el que se encuentran asociadas [8]:

1. ***Visual Servoing* o Control Visual.** Es una técnica que emplea información recibida por *feedback* que se obtiene a través de un sensor de visión para controlar el movimiento de un robot. Los paquetes más representativos son *vision_visp* o *ar_pose*.

2. **Percepción y reconocimiento de objetos.** Esta técnica se basa en el almacenamiento de modelos 3D o 2D de objetos para que sean reconocidos posteriormente y se establezca una relación entre los sistemas de coordenadas del objeto y la cámara. *RoboEarth* es el ejemplo más representativo.

3. **Odometría Visual.** Los algoritmos de odometría visual emplean imágenes del entorno para estimar el movimiento del robot asumiendo un entorno estático. *Viso2* es el paquete más empleado en estos casos.

2.1 Visual Servoing

Se denomina *Visual Servoing* al uso de *feedback* o retroalimentación de tipo visual como fuente de información para que dicho robot realice una tarea, comúnmente relacionada con su movimiento. Está basado en la idea de usar características visuales tales como puntos, líneas o regiones para, por ejemplo, permitir la alineación de un manipulador con un objeto a agarrar. Por tanto, la visión es parte de un sistema de control en el que provee de *feedback* acerca del estado del entorno en el que se encuentra el robot [9].

2.1.1 Definición

Como técnica, el *Visual Servoing* ha sido estudiado de diversas maneras a lo largo de tres décadas, desde que el Laboratorio Internacional del SRI (*Stanford Research Institute*) hablase de la técnica en uno de sus informes en 1979 [10], realizando simples tareas de emplazamiento de objetos, retransmitiendo la información en tiempo real. En términos de manipulación, una de las motivaciones principales de la incorporación de la visión en el lazo de control era la demanda de mayor flexibilidad en los sistemas robóticos. En el campo de la visión por computador se abren dos configuraciones fundamentales basadas en la posición del efector del robot con respecto a la posición de la cámara:

- ***Eye-in-hand***, o control de punto final en lazo cerrado, donde la cámara se encuentra unida al fin de la extremidad móvil, observando y retransmitiendo la posición relativa del objetivo con respecto a la cámara. El *Visual Servoing* aborda principalmente esta configuración.
- ***Hand to eye***, o control de punto final en lazo abierto, donde la cámara se encuentra fija con respecto al sistema de referencia "mundo" y observa el objetivo y el movimiento de la extremidad del robot.

En la figura 1 se muestran esquemáticamente estas dos configuraciones:

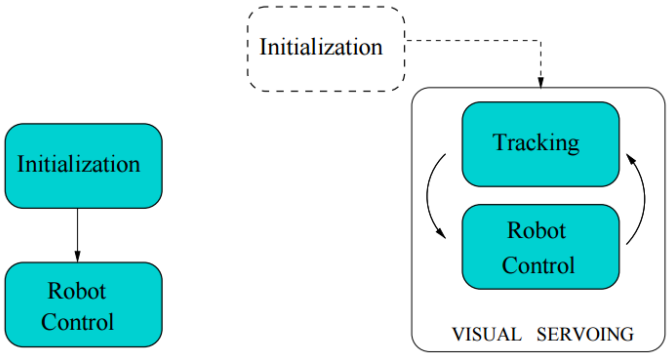


Figura 1. Diagrama de configuraciones de *cámara-efector*
Derecha) Configuración Eye-in-Hand
Izquierda) Configuración Hand to eye. [11]

Una tarea típica de *Visual Servoing* implica alguna forma de posicionamiento, como la alineación de una pinza con el objetivo a agarrar, o seguimiento de un objetivo móvil. En cualquiera de los casos la información de la imagen se emplea para medir el error entre la ubicación actual del robot y su referencia o ubicación deseada. La información que se emplea

para realizar la tarea puede ser de varios tipos lo que permite clasificar las técnicas de *Visual Servoing* en tres grupos [11].

- Basados en Imágenes 2D, en las que se logra encontrar el error comparando la imagen actual con la imágen de la posición deseada.
- *Pose-based*, o basadas en imágenes 3D, en las que la información que se recibe viene en términos de geometría tridimensional, es decir, obteniendo la posición y la orientación de las partes implicadas.
- Híbrido, lo que permite, combinando ambas técnicas, el reducir al máximo el error que se pueda cometer.

2.1.2 Proyectos Relevantes

2.1.2.1 Posicionamiento 3-D Autónomo de Instrumentos Quirúrgicos en Cirugía Laparoscópica

Se trata de un sistema de visión robótica que automáticamente recupera y posiciona instrumentos quirúrgicos durante operaciones laparoscópicas robotizadas. El instrumento se monta al final del efector de un robot quirúrgico que está controlado a través de *Visual Servoing.* El objetivo de la tarea automatizada es llevar un instrumento de forma segura desde una posición tridimensional desconocida u oculta a la deseada.

Unos diodos emisores de luz se encuentran unidos a la punta del instrumento y un soporte para el instrumento fabricado específicamente y equipado con fibra óptica proyecta un conjunto de puntos láser en la superficie de los órganos. Los marcadores ópticos son detectados en la imagen endoscópica y permiten localizar el instrumento con respecto a la escena. El instrumento es recuperado y centrado en la imagen empleando un algoritmo basado en detectar errores en detalles de las imágenes [12]. En la Figura 2 se muestra como está conformado este tipo de sistemas.

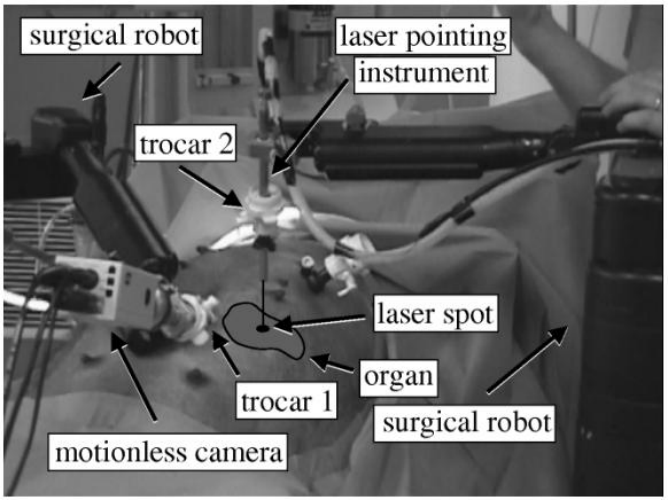


Figura 2. Configuración del sistema de Posicionamiento 3-D Autónomo de Instrumentos Quirúrgicos. [12]

2.1.2.2 Estabilización y Aterrizaje de *ARDrone*

En este caso, y atendiendo a la naturaleza del trabajo se expone esta aplicación, de mayor cotidianeidad y dedicada a un vehículo. Actualmente se están empleando una gran cantidad de bibliotecas de *Visual Servoing* para crear sistemas que permitan que el *ARDrone* de *Parrot* sea capaz de estabilizarse en una posición de manera autónoma [13].

Figura 3. *ARDrone* de *Parrot*. [Fuente: http://ardrone2.parrot.com/]

También se emplea para que el *dron* aterrice sobre un determinado marcador, y en caso de que lo pierda sea capaz de recuperar altura [14].

2.1.3 Relación con el problema propuesto

La idea general que ha motivado el desarrollo en estas últimas tres décadas del *Visual Servoing* es la de lograr minimizar el error especificado y obtener una referencia lo más fiable posible. Como técnica, se trata de un método ampliamente desarrollado en la actualidad y con una gran simpleza a la hora de ser implementado en un sistema al que se le quiera añadir visión por computador, ya que provee de una serie de datos muy concretos con relación al movimiento del objeto. El inconveniente es que es un método muy enfocado a la relación robot-objetivo, es decir, que no se referencia con el entorno como tal, por lo que depende de visualizar las marcas visuales para recibir la información y que el robot se oriente.

2.2 Percepción y Reconocimiento de Objetos

Un sistema de reconocimiento de objetos es capaz de encontrar objetos presentes en el mundo real a través de imágenes de dicho entorno, empleando modelos de los objetos conocidos a priori. La tarea como tal resulta sorprendentemente difícil, ya que un ser humano es capaz de esta tarea sin apenas esfuerzo y de manera casi instantánea. La descripción algorítmica de esta tarea para su implementación en máquinas ha sido muy laboriosa. Pese a ello se ha logrado desarrollar una serie de pasos para realizar el reconocimiento de objetos y se han podido establecer distintas técnicas que han sido empleadas para esta tarea en multitud de aplicaciones.

2.2.1 Definición

El problema asociado con el reconocimiento de objetos puede ser definido como un problema de clasificación basada en modelos de objetos conocidos. Es decir, de una imagen que contiene uno o más objetos de interés y un conjunto de etiquetas correspondientes a un grupo

de modelos conocidos para el sistema, éste debería ser capaz de asociar correctamente dichas etiquetas a regiones de la imagen analizada. Por tanto, se puede asociar estrechamente el problema del reconocimiento de objetos a un problema de segmentación: sin al menos un reconocimiento parcial de los objetos, la segmentación difícilmente será realizada y sin segmentación, el reconocimiento de objetos se torna imposible [15].

Un sistema de reconocimiento de objetos está formado por los siguientes componentes:

- **Una base de datos del modelo.** La información contenida en la base de datos dependerá del método de reconocimiento empleado.
- **Detector de características.** Aplica una serie de operadores a las imágenes e identifica localizaciones de características que ayuden a formar hipótesis.
- **Creador de hipótesis.** Empleando las características detectadas previamente se crean una serie de hipótesis que relacionan éstas con los registros de la base de datos.
- **Verificador de hipótesis.** Como es lógico, todas estas hipótesis han de ser verificadas, seleccionando las hipótesis que más coincidencias tengan.

La intercomunicación de estos elementos en un sistema de reconocimiento de objetos se muestra en la figura 4.

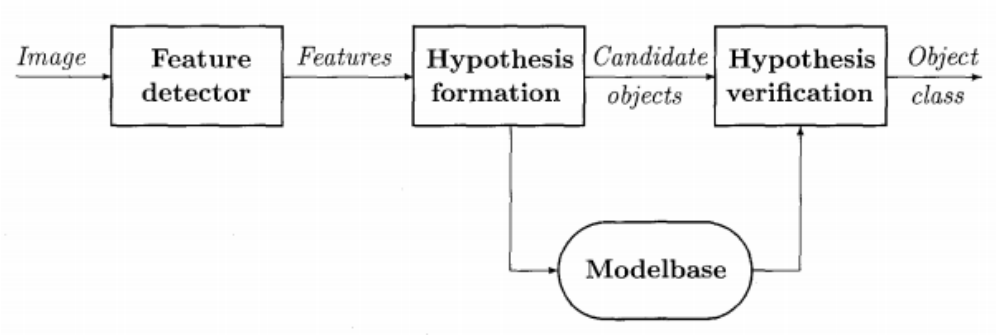


Figura 4. Componentes en un sistema de reconocimiento de objetos. [15]

Por tanto, para que un sistema de reconocimiento de objetos pueda realizar estos pasos mostrados arriba, se han de seleccionar las herramientas y métodos adecuados. Para ello se tienen que responder a una serie de cuestiones. Además de esto se han de incluir una serie de factores que también intervienen en el proceso, como la iluminación y los parámetros de la cámara y que por tanto añaden complejidad al reconocimiento de objetos:

- **Constancia de la escena**. Si las imágenes de muestra y los modelos almacenados poseen las mismas características.
- **Espacios de modelos de imágenes**. En algunos casos a partir de objetos tridimensionales se pueden considerar modelos de objetos bidimensionales. En esos casos los modelos se pueden crear a partir de parámetros bidimensionales.
- **Cantidad de objetos en la base de datos de modelos**. Cuando la base de datos no es muy amplia, la fase de hipótesis puede ser despreciada.

- **Numero de objetos en la imagen y posibilidad de oclusión**. El aumento de la cantidad de objetos en la imagen puede aumentar la probabilidad de que se produzca oclusión, que como resultado provoca que las características esperadas desaparezcan, lo que a su vez se ha de tener en cuenta en la fase de hipótesis.

2.2.2 Proyectos Relevantes

2.2.2.1 Detección de Objetos en Tiempo Real para Vehículos "Inteligentes"

En este trabajo se presenta un método de detección de siluetas de objetos basado en Transformadas de Distancia y describe su uso en vehículos con visión a bordo en tiempo real. El método emplea una jerarquía de plantillas para capturar una gran variedad de formas de objetos. Por un lado, *offline* se generan jerarquías en base a una serie de distribuciones de siluetas, empleando técnicas de optimización estocásticas. *Online,* encontrar coincidencias implica un enfoque de criba sobre la jerarquía de siluetas y sobre los parámetros de transformación [16]. El procesamiento al que se someten las imágenes se muestra en la figura 5.

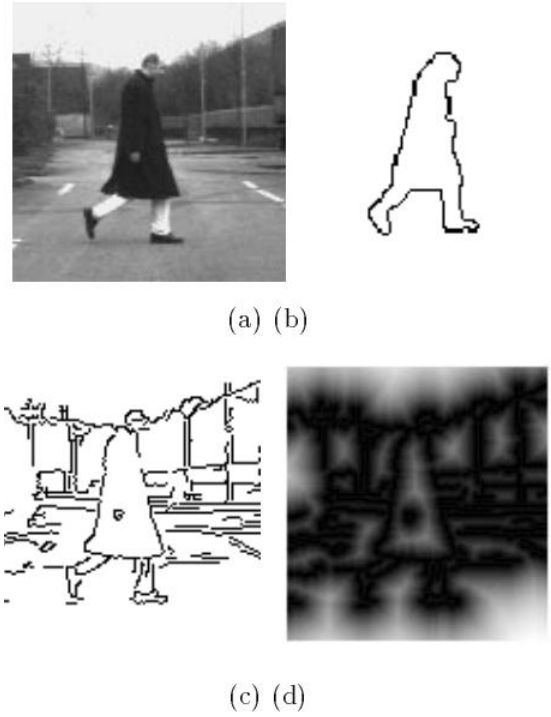


Figura 5. (a) imagen original, (b) plantilla, (c) aislamiento de bordes en la imagen, d) Imagen tras ser procesada por Transformadas de Distancia. [16]

2.2.3 Relación con el problema propuesto

Los sistemas de detección de objetos se perfilan como un método de gran versatilidad, a diferencia de otros métodos éste no requiere de marcas o señales específicas ya que permite almacenar siluetas en bases de datos. Precisamente las bases de datos son también su debilidad ya que el almacenamiento de dichos patrones y siluetas suele ser un método costoso si se quiere alcanzar cierto grado de fiabilidad. Además, en caso de que cambie la forma de algún objeto o se quiera añadir otro más, si la biblioteca está cerrada, es imposible.

2.3 Odometría Visual

Se conoce como Odometría Visual al proceso de determinar la posición y orientación de un robot analizando las imágenes obtenidas de una cámara asociada al mismo [17]. Se ha empleado en multitud de aplicaciones relacionadas con la robótica siendo la más conocida Los *Rovers* de Exploración en Marte [18].

2.3.1 Definición

El concepto odometría, en navegación se corresponde con el uso de la información obtenida a través del movimiento de unos actuadores para estimar el cambio de la posición en función del tiempo de aparatos como *encoders* rotatorios para medir rotaciones de ruedas. Mientras que es muy útil para múltiples vehículos sobre ruedas, este tipo de odometría es de difícil aplicación en robots, que poseen técnicas de movimiento que no son corrientes, como robots que poseen piernas. Además, la odometría sufre tradicionalmente de problemas de precisión, teniendo en cuenta la tendencia que tienen las ruedas a deslizarse sobre el suelo. Esto, provoca que las medidas odométricas se vuelvan poco fiables con el tiempo y se vaya acumulando una cantidad de error [19]. De este modo surge el concepto de odometría visual, que consiste en determinar la odometría equivalente empleando, en vez de sensores asociados a las ruedas, la información proveniente de las imágenes que emite una cámara asociada al robot para estimar la distancia recorrida.

Para que este método se realice adecuadamente se requiere asumir una serie de condiciones [17]:

- La suficiente iluminación en el entorno.
- El dominio de una escena estática sobre objetos móviles.
- La suficiente calidad de texturas para que se pueda percibir el movimiento.
- Que haya suficiente solapamiento de escenas entre *frames*.

Cumpliéndose esta serie de condiciones un sistema basado en odometría visual puede ser diseñado. Aunque se han hecho múltiples aproximaciones al concepto de odometría visual, todos se basan en una serie de fases:

1. Adquirir un conjunto de imágenes, ya sea a través de cámaras mono, estéreo o omnidireccionales.
2. Se ha de aplicar una serie de técnicas de procesamiento de imagen para liberarlas de distorsiones.
3. Detección de características. Al igual que ocurría con el reconocimiento de imágenes y el *Visual Servoing* se han de reconocer una serie de características que se emplearan como referencia a lo largo de cada *frame*, para construir lo que se denomina como campo de flujo óptico [20].
    a. Empleando correlación para establecer coincidencias entre imágenes.

b. Extracción de las características y correlación entre las mismas.

c. Construcción del campo de flujo óptico, empleando el método *Lucas-Kanade*.

4. Comprobar los vectores de flujo de campo para detectar errores de seguimiento potenciales y eliminar valores atípicos.

5. Estimación del movimiento de la cámara a través de este flujo óptico.

6. Periódicamente se repueblan los puntos de referencia para mantener la cobertura a través de la imagen.

2.3.2 Proyectos Relevantes

2.3.2.1 *NASA Mars Exploration Rover*

Los dos años de exploración de los *Rovers* de Exploración Espacial de Marte (*MER*) de la NASA ha demostrado satisfactoriamente la capacidad de la Odometría Visual en otro mundo por primera vez. La odometría visual permite que cada *Rover* tenga un conocimiento preciso de su posición, lo que le posibilita el detectar autónomamente y compensar cualquier deslizamiento imprevisto durante el trayecto. Esto habilita a los *Rovers* el viajar de manera segura y más efectiva en terrenos arenosos o con una gran pendiente, lo que ha incrementado la cantidad de información obtenida en su misión científica ya que se reducen ostensiblemente la cantidad de días requeridos para viajar entre las distintas zonas de interés.

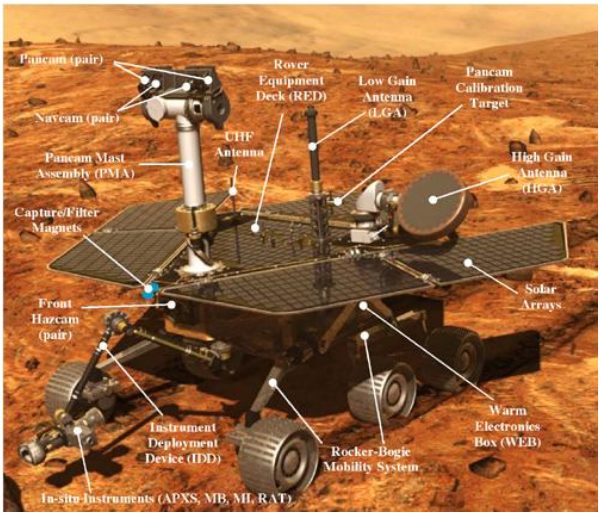


Figura 6. NASA *Mars Exploration Rover.* [18]

El sistema de odometría visual del *MER* incluye software de a bordo para comparar pares estéreo tomadas por las cámaras de navegación *FOV* dirigibles y montadas en un mástil con una inclinación de 45 grados (*NAVCAMs*). El sistema calcula una actualización de la posición y orientación del *rover* a través del seguimiento del movimiento de las características del terreno seleccionadas autónomamente, entre dos pares de imágenes estéreo con una resolución de 256x256. Se han demostrado un buen rendimiento del sistema con unos altos coeficientes de convergencia, detectando deslizamientos del 125%, y midiendo cambios de 2mm, incluso cuando se viaja con pendientes de 31 grados de inclinación [18].

2.3.3 Relación con el problema propuesto

La odometría visual constituye un método muy atractivo a la hora de realizar tareas que impliquen visión por computador. Entre las ventajas más llamativas se incluye que a diferencia de la odometría como tal, la odometría visual no se ve afectada por deslizamientos de la rueda ni condiciones adversas del terreno, en comparación las trayectorias son más precisas con un error en posición de entre el 0.1 y el 2% y en ambientes donde no hay referencias vía GPS adquiere una gran importancia. Como desventaja se puede citar que la cámara a emplear normalmente es estéreo, requiriendo de una mayor cantidad de información en el caso de usar una cámara mono y por tanto dificultando en gran medida su instalación [21].

2.4 Solución propuesta

Todas estas técnicas explicadas anteriormente demuestran la capacidad que posee actualmente la visión por computador para dotar de un gran y fiable flujo de datos al sistema en el que se quiera instalar. Prueban ser no sólo técnicas fiables a nivel de apoyo de un sistema principal, sino que podrían ser aplicados en cualquier caso como fuentes de información principal permitiendo el control total del sistema en cuestión. Es además algo remarcable que estas técnicas pueden ser perfectamente complementarias como es el caso del *Visual Servoing* y el reconocimiento de objetos, permitiendo referencias válidas no solo en balizas que se puedan emplazar, como suele ocurrir con el *Visual Servoing*, sino que acompañando esta técnica con una base de datos profunda de objetos del entorno en el que se mueva el robot, se puede lograr que éste se oriente a través de referencias presentes en su propio entorno, sin necesitar modificarlo.

En el caso que se presenta, que es básicamente el de acompañar con información puntual y auxiliar a un vehículo que controla su desplazamiento a través de odometría, cualquiera de estos métodos en incluso la combinación de varios de ellos puede llegar a ser válida como se ha explicado previamente. Pero se han de remarcar una serie de bases para priorizar cuál de estos métodos implementar. Se debe, por un lado, hacer cierto énfasis en la facilidad de implementación bajo las condiciones técnicas del caso, lo que lleva a remarcar, el tiempo de duración con el que se condiciona el proyecto. A esto además se ha de añadir que como sistema secundario de información acerca de la posición y orientación de un objeto se pueden establecer unos mínimos de fiabilidad, y se debe tener en cuenta el entorno en el que se pretende situar al vehículo. Por último, ya que es una primera implementación debe ser un método flexible, al que se le pueda ir dando mayor fiabilidad en caso de que se decida seguir ampliando este trabajo. El método que mejor equilibra facilidad de implementación, un mínimo de fiabilidad y suficiente flexibilidad es el *Visual Servoing,* siendo este el que se pretende emplear para el proyecto. *ROS* posee multitud de aplicaciones que emplean técnicas de visión por computador, de entre las que se encuentran clasificadas dentro de *Visual Servoing* se empleará *ar_pose*.

3. Descripción General

Teniendo en cuenta todo lo mencionado anteriormente se puede comenzar a acotar el proyecto.

El trabajo consiste en diseñar un sistema que sirva como apoyo a un sistema odométrico diseñado para controlar un carrito de golf. En este caso, este sistema es capaz de localizar donde se encuentra el carrito, pero cuanto mayor es el avance de éste, mayor es la incertidumbre que rodea a su posición. Esta área de incertidumbre se debe principalmente a que la odometría sufre tradicionalmente de problemas de precisión, a causa de la tendencia de las ruedas a patinar sobre el terreno. El ejecutable a diseñar servirá como una referencia que permita al carrito reiniciar la incertidumbre citada previamente cada vez que el carrito perciba visualmente unas balizas (en este caso marcas visuales) situadas en unos puntos estratégicos a lo largo del recorrido del vehículo. La figura 7 ilustra esquemáticamente el sistema descrito.

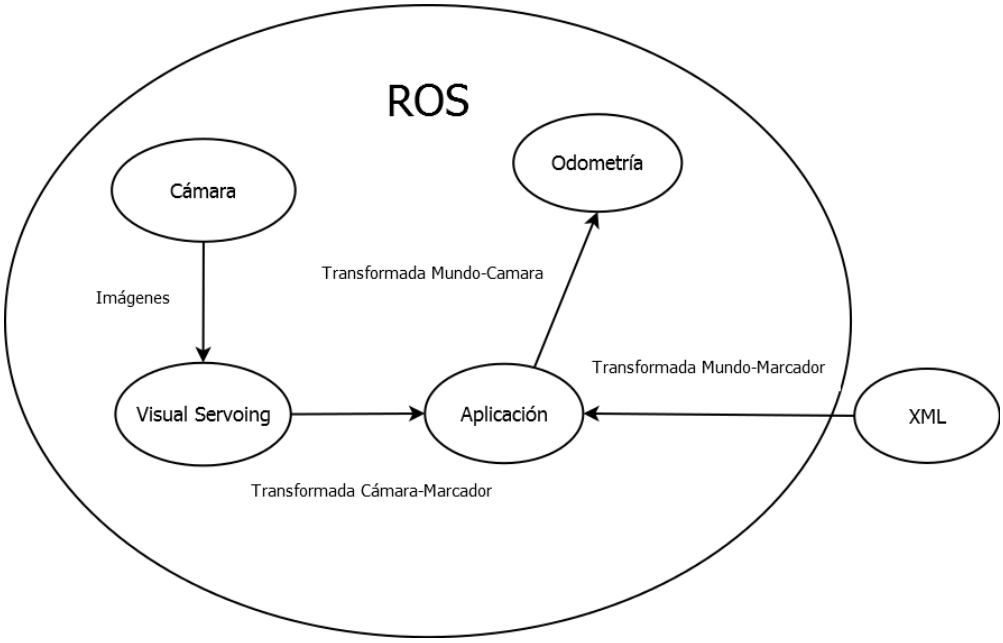


Figura 7. Esquema del Sistema.

Este sistema a diseñar estará formado por varios componentes que pertenecerán a distintas categorías dentro del trabajo en sí. Por tanto se divide el trabajo en tres grandes bloques:

- **Hardware**. Aunque no es la parte en la que se centre el proyecto, es cierto que el proyecto ha de estar condicionado por la cámara que se emplee para desarrollar el proyecto, además del ordenador en el que se trabajará y se diseñará la aplicación.
- **Software**. *ROS* es una plataforma de código abierto en el que las aplicaciones se comunican entre sí, emitiendo y recibiendo información, por lo que se ha de dejar claro que aplicaciones que pese a formar parte del proyecto ya existen y como se van integrando en el sistema completo.
- **Aplicación**. Aquí se engloba lo que se diseña originalmente para el proyecto, con qué se diseña, como se integra en el sistema y que resultados se obtienen.

4. Hardware

Pese a que el proyecto se encuentra centrado en un carrito de golf, el trabajo se desarrollara mayormente fuera de éste. Por un lado se trabajará, a la hora de programar en *ROS*, a través de un ordenador portátil cuyas prestaciones se explicarán más adelante. Por otro lado, con la cámara que se pretende trabajar es con la que viene integrada en el propio portátil. Esto es debido a que una de las motivaciones del proyecto es la de aprovechar al máximo los recursos disponibles.

4.1 Portátil ASUS K53S

Las restricciones que se pueden encontrar en este caso son el ser capaz de trabajar con *ROS*, cuyas exigencias no son excesivas, y por extensión poder trabajar con Ubuntu 14.04. Se pretende dedicar una partición con el espacio suficiente para instalar este sistema operativo. Las prestaciones que caracterizan el modelo y se encuentran relacionadas con los requerimientos tanto del sistema operativo como de *ROS* son [22]:

- **Procesador:** Intel® Core™ i7 2670QM/2630QM .
- **Disco Duro dedicado:** 20 G (proveniente de un Disco Duro de 640 G 5400rpm).
- **Memoria:** DDR3 1333 MHz SDRAM, 2 x SO-DIMM socket 8 G SDRAM.
- **Tarjeta Gráfica:** NVIDIA® GeForce® GT 540M con 2GB DDR3.

Figura 8. Portátil ASUS *K53S.*

4.2 Cámara

Para este proyecto se empleara la webcam integrada en el propio ordenador. Se trata de una cámara web fija de 0.3 megapixels, situada en el marco de la pantalla del ordenador, en la parte superior.

Lo cierto es que la cámara será lo que más restringa la elección del software durante el trabajo. Gran parte de la decisión de no emplear odometría visual en el proyecto radica en el hecho de que la odometría visual requiere de bastante información al trabajar con cámaras mono, como es el caso, dificultando ostensiblemente la implementación de la técnica.

Figura 9. Cámara del ASUS *K53S.*

5. Software

El objetivo principal del trabajo es construir un sistema que permita lograr un determinado objetivo. Este sistema a nivel de software estará formado por contenido original y por contenido ya creado, pero de código libre. Debido a esto, se ha de diferenciar claramente qué contenido del trabajo es original y qué no. En esta sección se explorara todo el trabajo que ha tenido que ver con software ya creado, y de qué manera se incorpora al trabajo.

Por un lado se encuentra *ROS*, el entorno de trabajo, que es la parte que caracteriza el proyecto, debido a su particular diseño modular. Está también la aplicación que se usará para hacer *Visual Servoing*, *ar_pose*; se explicará cómo funciona y por qué es la más adecuada para este trabajo. Se hablará también de las bibliotecas que se utilizarán en la fase de programación. Y aunque pueda no parecer importante, la incorporación de la cámara en el trabajo también requiere su propio ejecutable, y como es lógico ésta ha de ser calibrada. Además, para ver los resultados de posición y orientación que devolverá *ar_pose* de manera más intuitiva se empleará un simulador de entornos, conocido como *rviz*. Todo esto se ejecutará en un fichero de tipo *.launch* que se irá completando conforme se avance en el capítulo.

5.1 Entorno de Programación: ROS

ROS (*Robot Operating System*) es un entorno de trabajo flexible creado para diseñar y ejecutar aplicaciones de robótica. Es un conjunto de herramientas, bibliotecas y convenciones que buscan simplificar la tarea de crear conductas para robots complejas y robustas a través de una amplia variedad de plataformas relacionadas con la robótica [4].

Figura 10. Logotipo *ROS Indigo*. [Fuente: ros.org]

5.1.1 Conceptos Principales

A nivel conceptual, el sistema de construcción de *ROS* en su versión *Indigo*, *catkin*, se divide en tres niveles diferentes: Sistema de Archivos, Grafica de Computación (*Computation Graph*) y la Comunidad [23]. Ya que los dos primeros niveles intervienen directamente con la comprensión del trabajo como conjunto, se pasa a describirlos con mayor detalle.

5.1.1.1 Sistema de Archivos

Los conceptos relacionados con el Sistema de Archivos cubren principalmente los recursos relacionados con *ROS* que se encuentran en el Disco Duro:

- **Paquetes (*package*):** Son la unidad principal de organización de software con la que trabaja *ROS*. Un paquete puede contener aplicaciones (llamadas nodos), bibliotecas con dependencia a *ROS*, conjuntos de datos, ficheros de configuración, ficheros ejecutables o cualquier cosa que sea practico organizar junto a lo anterior. Los paquetes son la unidad atómica de construcción y liberación presente en *ROS*.

- **Manifiestos de Paquetes (*package manifest*):** El fichero *package.xml* presente en un paquete, proporciona información relacionada con el paquete en el que se encuentra contenido, entre la que se encuentra el nombre, la descripción, la información relacionada con su licencia o sus dependencias con respecto a otros paquetes.

- **Repositorios (*repository*):** Una colección de paquetes que comparten un sistema *VCS* (*Version Control System*) común. Los paquetes que comparten un *VCS* comparten la misma versión y pueden ser liberados al mismo tiempo.

- **Tipos de Mensajes (*msg*):** Las descripciones de los mensajes, almacenadas en una dirección similar a *my_package/msg/MyMessageType.msg*, definen como se encuentra estructurada la información contenida en un mensaje (explicado más adelante) que se envía en *ROS*.

- **Tipos de Servicios (*srv*):** Las descripciones de los servicios, guardadas en my_package/srv/MyServiceType.srv, definen las estructuras de datos de petición y respuesta para servicios (explicado más adelante) en *ROS*.

5.1.1.2 Gráfica de Computación

La Grafica de Computación (*Computation Graph*) se define como la red *peer-to-peer* de procesos de *ROS* que se encuentran procesando información al mismo tiempo. Los conceptos relacionados en este ámbito con mayor relevancia son los nodos (*nodes*), el Maestro (*Master*), el Servidor de Parámetros (*Parameter Server*), mensajes (*messages*), servicios (*services*), temas (*Topics*), y bolsas (*bags*), todos ellos proporcionan información a la Gráfica de diferentes maneras.

- **Nodos (*Nodes*):** Son ejecutables que realizan diversos procesos. *ROS* está diseñado para ser modular a la mayor escala posible; por lo que el sistema de control de un robot conjugará varios nodos. En este caso, por ejemplo, uno de los nodos se encargará de poner en funcionamiento la cámara, otro averiguará la posición relativa

de la cámara con respecto a una baliza y otro calculará la posición global de la cámara, enviando a su vez esta información. Un nodo de *ROS* se escribe empleando bibliotecas de cliente de *ROS*, como *roscpp* (en *c++*) o *rospy* (empleando *pyton*)

- **Maestro (*Master*):** El *ROS Master* proporciona el registro y búsqueda de nombres al resto de la Grafica de Computación. El maestro permite que los nodos se encuentren unos a otros, intercambien mensajes o pidan servicios.

- **Servidor de Parámetros (*Parameter Server*)**: El Servidor de Parámetros permite guardar información bajo llave en una localización principal. Actualmente forma parte del maestro.

- **Mensajes (*Messages*)**: Los nodos se comunican entre sí intercambiando mensajes. Un mensaje es una estructura de datos simple, que almacena una serie de variables (las variables estándar como *integers, floats o bools* están incluidas) u objetos de una clase.

- **Temas (*Topics*)**: Los mensajes son enviados empleando semántica de tipo publicación/subscripción. Un nodo envía un mensaje publicándolo en un determinado *topic* o tema. Un tema es la denominación que se emplea para identificar el contenido del mensaje. Un nodo que esté interesado en información de un determinado tipo, se subscribirá al tema correspondiente. Para un solo tema puede haber múltiples publicadores y subscriptores, así como un solo nodo puede publicar y subscribirse a varios temas al mismo tiempo. Siguiendo la idea de separar la consumición de información de su emisión, los nodos publicadores no son conscientes de la existencia de los subscriptores y viceversa.

- **Servicios (*Services*)**: A pesar de que el sistema de publicación y subscripción constituye un ejemplo de comunicación muy flexible, no resulta del todo apropiado cuando se busca una interacción de tipo petición/respuesta, normalmente requeridos en sistemas distribuidos. En *ROS*, esta interacción se realiza a través de *services* o servicios, que se definen empleando un par de estructuras de mensajes: una se emplea para la petición y otra para la respuesta correspondiente. Un nodo ofrece un determinado servicio bajo un nombre y un cliente emplea dicho servicio enviando un mensaje de petición y esperando una respuesta. La mayor parte de bibliotecas de *ROS* ofrecen esta interacción con el programador como si se tratara de una llamada a un procedimiento remoto.

- **Bolsas (*Bags*)**: Las *bags* o bolsas son un formato dedicado a guardar y reproducir información relacionada con mensajes en *ROS*. Las *bags* son un mecanismo de gran relevancia para almacenar información, como por ejemplo información de un sensor, que suele ser relativamente difícil de guardar pero indispensable a la hora de desarrollar y probar algoritmos.

El *Master* de *ROS* actúa como un servicio de nomenclatura en la Grafica de Computación de *ROS*. Almacena temas y provee de datos de registro para los nodos. Los nodos se comunican con el *Master* para reportar acerca de sus datos de registro. Al comunicarse con el *Master*

reciben también información acerca de otros nodos registrados, lo que permite establecer comunicaciones correctamente. El *Master* también permite informar a los nodos cuando los datos de registro cambien, de modo que los nodos que ya se encuentran funcionando puedan establecer comunicaciones cuando un nuevo nodo se ejecuta.

Aun así los nodos se conectan entre sí directamente, el *Master* únicamente se encarga de darles información de búsqueda, de un modo parecido a como lo haría un servidor *DNS*. Los nodos que se subscriben a un tema pedirán conexión a los nodos que la publican en dicho tema, y establecerán una conexión a través de protocolo. El protocolo de conexión más empleado en *ROS* es el *TCPROS*, basado en protocolos *TCP/IP* estándar.

Este tipo de arquitectura permite operaciones desvinculadas, donde los nombres son los medios primarios a través de los cuales se pueden construir mayores y más complejos sistemas. Los nombres tienen una gran importancia dentro de *ROS*: los nodos, temas, servicios y parámetros poseen nombres. Cada biblioteca de cliente de *ROS* soporta el remapeado de nombre a través de la línea de comando, lo que implica que un programa que se encuentra compilado puede ser reconfigurado mientras que se está ejecutando para trabajar en una topología diferente en la Grafica de Computación.

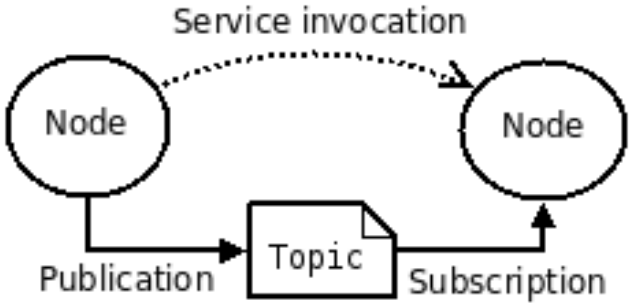


Figura 11. Esquemático de la Gráfica de Computación. [Fuente http://wiki.ros.org/*ROS*/Concepts]

5.2 Herramientas de Programación

Una vez que se ha creado el paquete, se puede comenzar a programar, pero antes de ello se ha de saber cómo se va a hacer y en qué idioma. En esta sección se profundizará en el lenguaje de programación que se va a emplear en el paquete, así como en las principales bibliotecas que se van a emplear para programar, ya sean externas o propias de *ROS*.

5.2.1 Lenguaje de Programación: *C++*

C++ es un lenguaje de programación diseñado a mediados de los años 1980 por Bjarne Stroustrup [24]. Fue creado con la intención de ampliar el lenguaje de programación *C* con mecanismos que permiten la manipulación de objetos. En ese sentido, desde el punto de vista de los lenguajes orientados a objetos, el *C++* es un lenguaje híbrido. *C++* es un lenguaje multiparadigma, para programación estructurada y programación orientada a objetos, muy útil en aplicaciones constituidas por varios módulos, o por varias clases.

Como se ha explicado previamente, *ROS* posee dos bibliotecas de cliente: *rospy*, relacionada con *Pyton*, y *roscpp*, basada en *C++*. Para este trabajo, basándose en los conocimientos de programación del alumno, se va a emplear *roscpp*, por lo que se programará en *C++.* Para ello se incluye *roscpp* como dependencia en el *package.xml* y en el *CMakeLists.txt*

5.2.1.1 Clases

Los objetos en *C++* son abstraídos mediante una clase. Según el paradigma de la programación orientada a objetos un objeto consta de:

1. Identidad, que lo diferencia de otros objetos (Nombre que llevara la clase a la que pertenece dicho objeto).
2. Métodos o funciones miembro
3. Atributos o variables miembro

5.2.1.2 Bibliotecas de *C++*

Los lenguajes de programación suelen tener una serie de bibliotecas de funciones integradas para la manipulación de datos a nivel más básico. En *C++*, además de poder usar las bibliotecas de *C*, puede usar la nativa *STL* (*Standard Template Library*), propia del lenguaje. *C++* proporciona una serie de plantillas (*templates*) que permiten efectuar operaciones de almacén de datos y procesos de entrada/salida. Las clases *basic_ostream* y *basic_istream*, proporcionan la entrada y salida estándar de datos (teclado/pantalla).

5.2.1.3 Compilación y Desarrollo en C++: *QtCreator*

Para este proyecto se empleará como IDE (Entorno de Desarrollo Integrado) *QtCreator.* De entre las ventajas que ofrece se citan las siguientes [25]:

- Editor de código con soporte para C+*, QML y ECMAscript*
- Herramientas para la rápida navegación del código

- Resaltado de sintaxis y auto-completado de código
- Control estático de código y estilo a medida que se escribe
- Soporte para *refactoring* de código
- Ayuda sensitiva al contexto
- Plegado de código (*code folding*)
- Paréntesis coincidentes y modos de selección

Pero la característica más relevante y la que determina su elección, es la alta compatibilidad que tiene con *ROS*. *QtCreator* no solo trabaja con *QtProjects*, sino que es totalmente compatible con los ficheros *CMakeLists.txt* como el que se encuentra en el fichero *src* del *workspace*. A través de la carga del archivo, *QT* es capaz de gestionar y construir todo el *workspace* sin necesidad de ejecutar el comando de construcción básico de *ROS catkin_make*.

5.2.2 Biblioteca externa: *TinyXml*

TinyXML es, en esencia, un analizador de documentos *XML*, construyendo desde éste un Modelo de Objeto de Documento (*DOM*) que puede ser leído, modificado y guardado [26].

Figura 12. Logo del software *TinyXml.*

5.2.2.1 XML

El formato *XML*, cuyas siglas representan Lenguaje de Etiquetado Extensible (*eXtensible Markup Language*), permite al usuario crear etiquetas en un documento. Donde *HTML* hace un muy buen trabajo etiquetando documentos provenientes de navegadores, *XML* permite definir cualquier tipo de etiqueta en un documento, por ejemplo, un documento que describa una lista de cosas por hacer para una aplicación destinada a la organización. *XML* es un formato muy cómodo y estructurado; cualquier formato destinado a almacenar información de una aplicación puede ser sustituido por *XML*.

5.2.2.2 Funcionamiento

Hay múltiples maneras de acceder e interactuar con la información contenida en un archivo *XML*. *TinyXML* emplea, como se ha indicado antes, *DOM*, lo que significa que la información en el archivo se puede volcar en un objeto de *C++* que pueden ser navegados y manipulados, y

posteriormente escritos en un disco o cualquier canal de salida. También se puede construir un documento *XML* desde un conjunto de objetos en *C++* y posteriormente ser escrito sobre un canal de salida.

TinyXML está diseñado para ser fácil y rápido de aprender. Está compuesto por dos archivos *.h* y cuatro *.cpp*. La manera de incorporarlos al trabajo es simple: se añaden directamente a las correspondientes carpetas *include* y *src*, siendo referenciados en el resto de archivos incluido el *CMakeLists.txt*, en el que se incluyen los *.cpp* como una biblioteca más.

5.2.2.3 Licencia

TinyXML está lanzado bajo la licencia *ZLib*, lo que implica que puede ser usado tanto en código libre como en código destinado a uso comercial. Los detalles de cada licencia se encuentran en la parte superior de cada archivo *.cpp*.

5.2.2.4 Limitaciones

TinyXML intenta ser un analizador flexible, pero con una salida en formato *XML* totalmente correcta y compatible. *TinyXML* debería compilar en cualquier sistema en *C++* razonablemente compatible. No depende de excepciones o *RTTI* (*Run-Time Type Information*). Puede ser compilado con o sin soporte *STL* (*Standard Template Library*). *TinyXML* es totalmente compatible con la codificación *UTF-8*, y las primeras 64k entidades de caracteres.

5.2.3 Bibliotecas de *ROS*

5.2.3.1 Biblioteca *ROS*

Para emplear todos los objetos que pertenezcan a ROS como tal y que se involucren con su sistema de comunicaciones, se tiene que incluir en primer lugar la biblioteca principal de ROS, indicando en el *header* en el que se van a incluir las bibliotecas la siguiente línea de código.

#include "ros/ros.h"

5.2.3.1 Biblioteca *tf*

Tf es un paquete de *ROS* que permite al usuario seguir a un conjunto de ejes de coordenadas a lo largo del tiempo. *Tf* mantiene las relaciones entre ejes de coordenadas en una estructura en forma de árboles, y permite al usuario transformar puntos, vectores, etc, entre dos ejes de coordenadas en cualquier momento [27].

Tf puede operar en un sistema distribuido. Esto significa que la información sobre los ejes de coordenadas de un robot está disponible para todos los componentes de ROS en cualquier equipo del sistema. No hay un servidor central de información de transformadas. Esta funcionalidad aplicada al trabajo permitirá comunicar los resultados de modo que puedan ser recibidos por el visualizador de entornos 3-D *rviz* y visualizados en tiempo real, facilitando la verificación de resultados en cada fase del desarrollo de la aplicación.

Pero la importancia que cobra *tf* en el trabajo no es como paquete, sino como biblioteca, por esa razón se le incluye en esta sección. *Tf* posee las clases que permiten trabajar con la

cinemática de un robot, y además posee métodos que facilitan los cálculos y conversiones en este campo. Algunas de las clases que contiene *tf* se pueden ver en la siguiente tabla.

Type	tf
Quaternion	tf::Quaternion
Rot. Matrix	tf::Matrix3x3
Vector	tf::Vector3
Point	tf::Point
Pose	tf::Pose
Transform	tf::Transform

Tabla 1. Tipos de datos presentes en *tf*.

Para incluir todas estas herramientas, en el *header* principal se incluye la siguiente línea de código al inicio.

#include <tf/transform_broadcaster.h>

5.3 Empleando una cámara con ROS: gscam

Una vez que *ROS* está completamente instalado y funcional en el ordenador se pasa a ir encontrando los paquetes necesarios para ir dando forma al trabajo. El primer paquete que hay que encontrar y del que dependerá el resto, ya que este es un trabajo de visión por computador, es el que permita trabajar con la cámara.

ROS tiene una gran biblioteca de drivers para trabajar con cámaras de múltiples resoluciones. Como en este caso se va a trabajar con la cámara integrada en el ordenador, que es de baja resolución, se necesita un driver que se centre en este concepto y que por supuesto sea compatible con cámaras web. También se necesitará que el driver sea compatible con el resto de paquetes que se irán empleando conforme se avance en el proyecto, enviando los mensajes adecuados bajo temas compatibles.

El paquete que cumple todos estos requerimientos y que por tanto será el seleccionado para este trabajo, permitiendo además múltiples opciones de procesamiento de imagen, es *gscam*.

5.3.1 *API* de *ROS*

Los *topics* que publica este paquete en *ROS* son [28]:

- ***image_raw (sensor_msgs/Image)***

  La imagen recogida por la cámara, sin procesar.

- ***camera_info (sensor_msgs/CameraInfo)***

  Contiene la calibración de la cámara (en caso de estarlo) e información extra acerca de la configuración de la cámara.

Los servicios que ofrece este paquete son:

- ***set_camera_info (sensor_msgs/SetCameraInfo)***

  Introduce la información de la cámara deseada (nombre del *frame* de la Transformada, parámetros de calibración, etc.).

5.3.2 Aplicación en el proyecto

En el trabajo, *gscam* será el primer nodo a ejecutar, ya que es el único que no se necesita subscribir a ningún otro. Para ejecutar *gscam*, se va a crear un fichero de tipo *.launch* que iremos completando con el resto de nodos que se van a ir describir a lo largo de este capítulo. Los ficheros *.launch* se ejecutan mediante el comando de *ROS roslaunch* y son ficheros *.xml* donde se indican los nodos a ejecutar, con que nombre se ejecutan y se asignan además valores a los parámetros que dichos nodos tienen disponibles.

En primer lugar se crea un argumento con el nombre de la cámara y posteriormente se inicializa el nodo en sí. Cuando se inicializa el paquete se indican datos en lo referente a la resolución, *framerate,* etc. Y se indica cual es el nombre de la cámara, como se llama el archivo de calibración y donde se encuentra. Por último se remapea el *topic* a través del cual este nodo va a publicar, para que sea más fácil de encontrar por parte de otros nodos.

Como se ha explicado, *gscam* constituye el primer elemento que se va a agregar a la Gráfica de Computación. Si se ejecuta la herramienta *rqt_graph*, que sirve para mostrarla esquemáticamente, se puede ver como *gscam* se encuentra publicando en ROS, imagen mostrada en la figura 13.

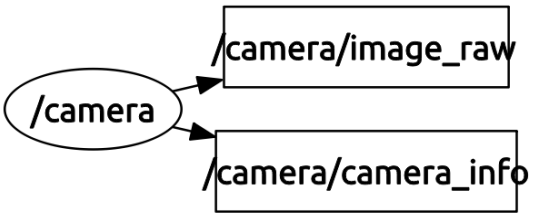


Figura 13. *Gscam* en la Gráfica de Computación.

5.4 Visual Servoing con ROS: ar_pose

El paquete *ar_pose* es un *wrapper* (una adaptación de una biblioteca a un entorno, en este caso ROS) de el conocido software de AR (*Augmented Reality*) *ArtoolKit* desarrollado por el Dr. Hirokazu Kato [29]. Se trata de una biblioteca para diseñar en esencia aplicaciones de realidad aumentada. Estas son aplicaciones que superponen animaciones virtuales sobre imágenes reales.

La cuestión es, ¿por qué se emplea una aplicación de Realidad Aumentada en un proyecto que no está basado o no pretende emplear este tipo de técnica? Como se explicará más adelante

cuando se profundice en los principios de funcionamiento, a la hora de diseñar una aplicación de realidad aumentada, hasta cierto punto se están compartiendo procesos con aplicaciones destinadas al *Visual Servoing*. A la hora de situar la imagen virtual en su correspondiente posición y orientación se necesita un *feedback* de cómo se encuentra situada la superficie sobre la que se quiere poner la imagen virtual, al igual que en el *Visual Servoing*. Además, para seleccionar qué imagen se desea superponer, este software es capaz de diferenciar entre imágenes, e incluso diseñar y personalizar éstas. El paquete de *Visual Servoing* de *ROS vision_visp* [30], a diferencia de *ar_pose,* no provee de dicha información, lo que dificulta el asociar y diferenciar entre códigos, lo cual es crucial en este proyecto.

5.4.1 Principios de funcionamiento

ARToolKit emplea algoritmos de visión por computador para realizar tareas de Realidad Aumentada. Las bibliotecas de detección de video de *ARToolKit* calculan la posición real de la cámara y su orientación con respecto a marcadores físicos en tiempo real [31]. El software sigue los pasos que se muestran en la siguiente imagen:

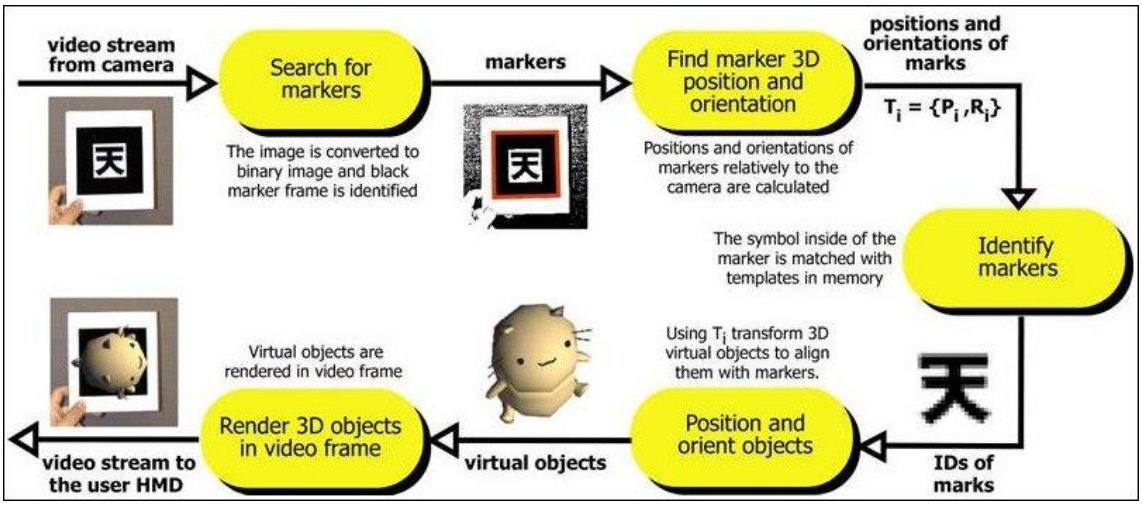


Figura 14. Patrón empleado en la calibración de la cámara. [Fuente: http://www.hitl.washington.edu/artoolkit/documentation/userarwork.htm]

Del esquemático de la figura 14, los pasos que se corresponden con el paquete *ar_pose* son los siguientes

1. La cámara graba video del mundo real y lo envía al ordenador.
2. El software del ordenador procesa cada *frame* del video y busca las formas cuadradas.
3. Si se encuentra un cuadrado el software emplea geometría para estimar la posición de la cámara en función del cuadrado detectado.
4. Se envía dicha estimación con una covarianza e indicando cual es la identidad de dicho marcador.

En figura 15 se observan los pasos que sigue el software para procesar la imagen y posteriormente estimar la posición y orientación de la marcador con respecto a la cámara.

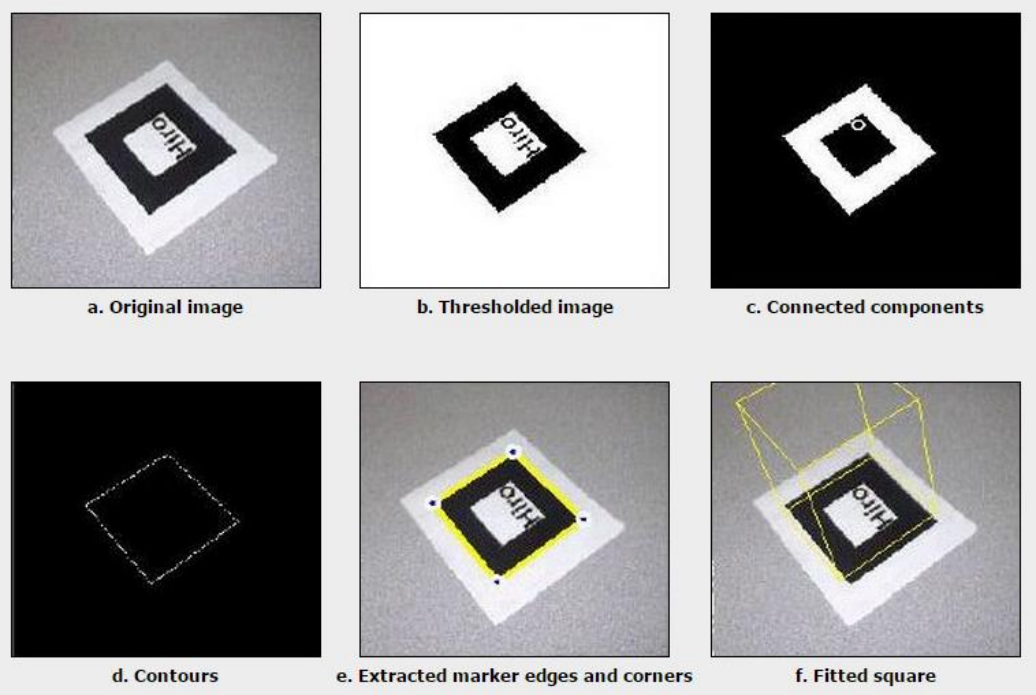


Figura 15. Procesamiento al que se somete a la imagen con marcador. [Fuente: http://www.hitl.washington.edu/artoolkit/documentation/vision.htm]

5.4.2 *API* de *ROS*

Ar_pose posee dos nodos distintos a ejecutar, *ar_single* y *ar_multi*. Debido a que la intención es que se puedan detectar múltiples balizas al mismo tiempo y gestionarlas en el paquete a diseñar del proyecto, se empleará *ar_multi*.

Los *topics* a los que se subscribe son [32]:

- ***/camera/image_raw (sensor_msgs/Image)***

  Las imágenes recibidas por este *topic* se pasan al software de *ARToolKit* para realizar el análisis descrito.

- ***/camera/camera_info (sensor_msgs/CameraInfo)***

  Contiene la calibración de la cámara (en caso de estarlo) e información extra acerca de la configuración de la cámara. Esta información se pasa al software de *ARToolKit.*

Los *topics* que publica *ar_multi* son los siguientes:

- ***ar_pose_marker***

  Un *array* de marcadores, cada uno con la información concerniente a la posición y la orientación del marcador con respecto a la cámara.

- ***visualization_marker***

  Un marcador visual que se envía para *rviz* (explicado más adelante)

Los parámetros que posee *ar_multi* son:

- ***marker_pattern_list (string, default: "data/object_4x4")***

  El archivo de *ARToolKit* donde se describe cada patrón.

- ***threshold (double, default: 100)***

  El umbral que se pasa a *ARToolKit*.

- ***publish_visual_markers (bool, default: true)***

  Opción que permite publicar los marcadores visuales para *rviz*.

- ***publish_tf (bool, default: true)***

  Permite emitir las transformadas.

Y por último, las transformadas que envía el ejecutable son las siguientes:

- ***camera_frame_id → marker_frames***

  Contiene la calibración de la cámara (en caso de estarlo) e información extra acerca de la configuración de la cámara.

5.4.3 Aplicación en el proyecto

En el sistema, *ar_pose* una vez ejecutado se podrá subscribir al *topic* publicado por *gscam*. Para que lo haga correctamente se incluirá la ejecución del nodo en el fichero *rviz_launcher.launch.*

Lo que se va a realizar en el ejecutable es inicializar el nodo *ar_multi,* y los parámetros que se especifican son el fichero donde se encuentran los patrones (“marker_pattern_list”) que será “object_4x4”.

Los resultados de su correcta subscripción se pueden observar ejecutando el nodo *rqt_graph*, el cual muestra esquemáticamente las comunicaciones que se producen en *ROS*. En la figura 16 se muestran los resultados de la ejecución de *rqt_graph*:

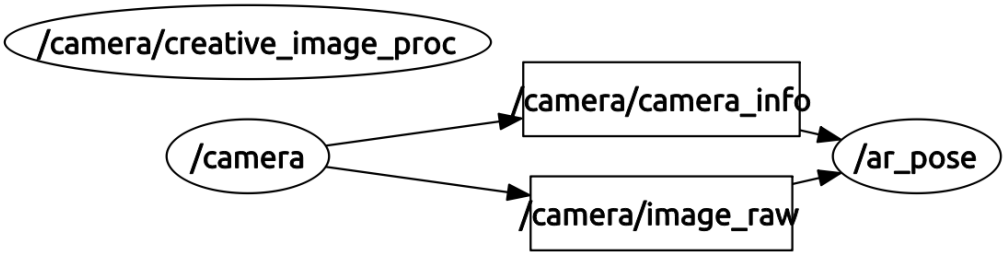


Figura 16. Interacción entre *ar_pose* y *gscam* en la Gráfica de Computación.

Como se puede observar el nodo *gscam*, a través de nombre *cámara* publica mensajes a través de dos *topics* (*/camera_info* y */image_raw*) y *ar_pose* se subscribe satisfactoriamente a ambos. Cuando *ar_pose* detecte un marcador comenzará a publicar la información correspondiente a la posición y orientación del marcador con respecto a la cámara.

5.4.4 Limitaciones

A la hora de implementar y tomar la decisión final de incorporar esta aplicación al proyecto, se han evaluado cuáles son sus limitaciones, así como su respuesta y grado de error en diversas circunstancias. En primer lugar, la limitación más importante, es que para que el software funcione correctamente, el marcador se ha de ver al completo. Este concepto es el más lógico y es casi una conclusión teniendo en cuenta la información expuesta hasta ahora.

También hay que tener en cuenta los problemas de rango [33]. Cuanto mayor sea el marcador, mayor será la distancia a la cual la cámara podrá detectarlo. En la tabla 2 se muestran rangos máximos típicos para marcadores cuadrados de distintos tamaños y en la figura 17 se encuentran dispuestos en una gráfica. Estos resultados se obtienen situando los marcadores enfrente de la cámara y alejándolo en perpendicular a ésta hasta que se deja de recoger la información.

Pattern Size (inches)	**Usable Range** (inches)
2.75	16
3.50	25
4.25	34
7.37	50

Tabla 2. Patrón empleado en la calibración de la cámara. [Fuente: http://www.hitl.washington.edu/artoolkit/documentation/userarwork.htm]

Figura 17. Detección de patrón en función de la distancia. [Fuente: http://www.hitl.washington.edu/artoolkit/documentation/benchmark.htm]

La gráfica de la figura 18 muestra el error en distancia en función del rango de detección del patrón.

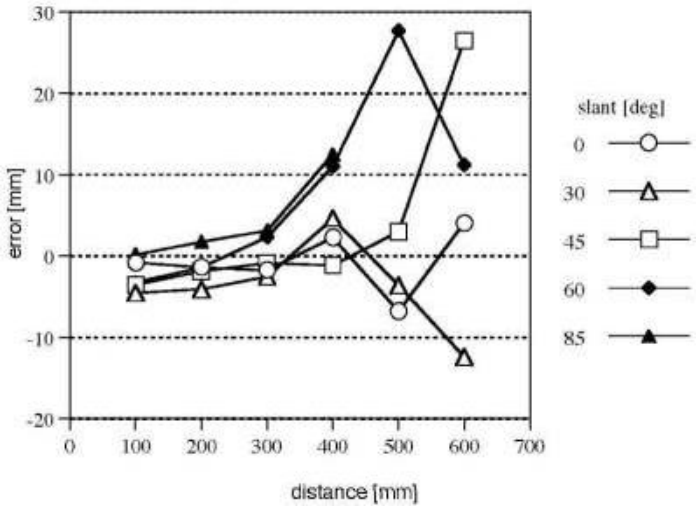


Figura 18. Error en distancia en función de la distancia de detección. [Fuente: http://www.hitl.washington.edu/artoolkit/documentation/benchmark.htm]

La complejidad del patrón también es un factor importante. Cuanto mayor sea el patrón mejor. Por ejemplo, los patrones que poseen grandes regiones blancas y negras ofrecen una mayor detección. Modificando el patrón de 4.25 pulgadas, haciéndolo más complejo, puede llegar a reducir el rango de detección de 34 a 15 pulgadas.

La orientación del marcador con respecto a la cámara también juega un papel importante en la detección. Cuanto más inclinados estén los marcadores, mayor dificultad habrá para detectar su centro, de modo que la detección se vuelve más difícil y poco fiable. La figura 19 muestra los valores de inclinación detectados en función de la distancia del marcador.

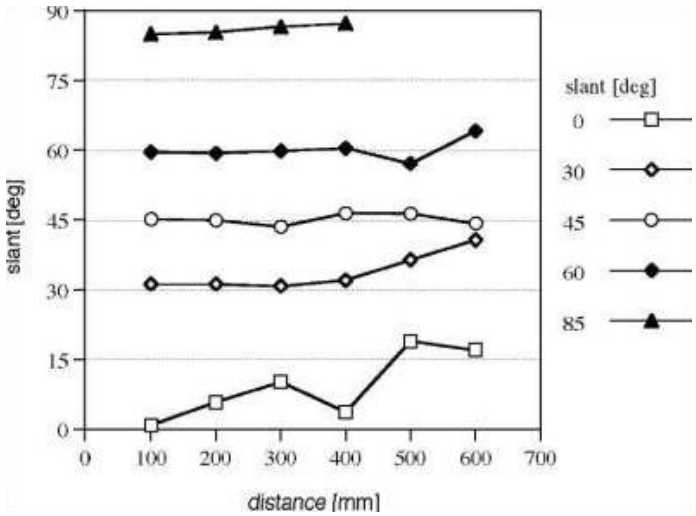


Figura 19. Ángulo en función de la distancia de detección. [Fuente: http://www.hitl.washington.edu/artoolkit/documentation/benchmark.htm]

Por último, se ha de destacar que la detección también se verá afectada por las condiciones de iluminación del entorno y la reflexión que se produzca sobre la superficie en la que el marcador esté impreso. Si la superficie es excesivamente reflectante, en condiciones de alta iluminación el patrón puede volverse muy difícil de detectar.

5.5 Entorno de visualización 3-D: rviz

El último paquete de software con el que se va a trabajar es con *rviz*. Su papel en el trabajo, como se profundizará en adelante, es paralelo al proyecto, siendo una herramienta que permitirá trasladar la información en forma numérica a un entorno gráfico, lo que hará de la verificación de resultados algo más intuitivo.

5.5.1 Descripción

Rviz es un entorno de visualización en 3-D en *ROS*. *Rviz* permite conocer lo que un robot está viendo, pensando o haciendo en tiempo real. Desarrollar un robot puede ser muy difícil si no se sabe qué es lo que el robot está percibiendo en su entorno. Depurar código mirando únicamente la consola ya es difícil en 2-D, y a menos que se tenga un conocimiento profundo de cuaterniones y ejes de coordenadas y una gran capacidad de abstracción, depurar en 3-D puede ser una tarea incluso más tediosa. *Rviz* permite al desarrollador mirar a través de los "ojos" del robot, independientemente de que dichos "ojos" puedan ser cámaras, láseres o *encoders*.

Principalmente, hay dos maneras de introducir información en el mundo de *rviz*:

- **Información proveniente de sensores**. *Rviz* comprende información proveniente tanto de escáneres láser, nubes de puntos, cámaras o ejes de coordenadas y tiene métodos específicos de mostrar los mismos, en función de las preferencias del usuario.

- **Marcadores de visualización.** Permiten al programador enviar la información en forma de cubos, flechas, etc.

La combinación de la información obtenida a través de la combinación de los sensores y los marcadores personalizados es lo que hace de *rviz* una herramienta muy potente cuando se trata de desarrollar las capacidades de un robot. Esta combinación es lo que se denomina el *stack* de navegación de *ROS* y muestra información de obstáculos, una rejilla vóxel 3D y un mapa topológico. Los marcadores de *rviz* actualmente se emplean en proyectos que implican detección de objetos o calibración. La figura 20 muestra un ejemplo del interfaz de *rviz*:

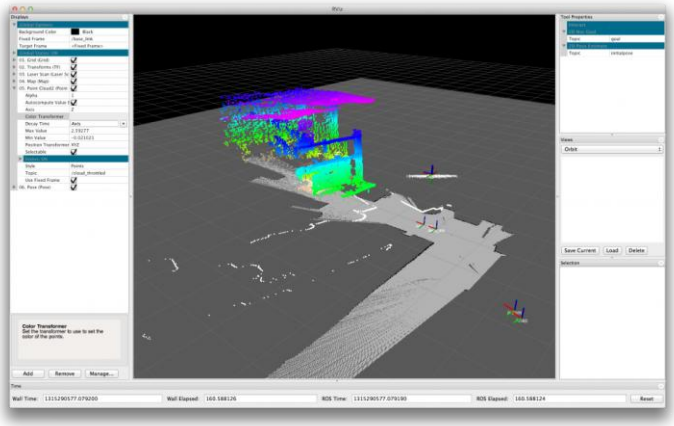
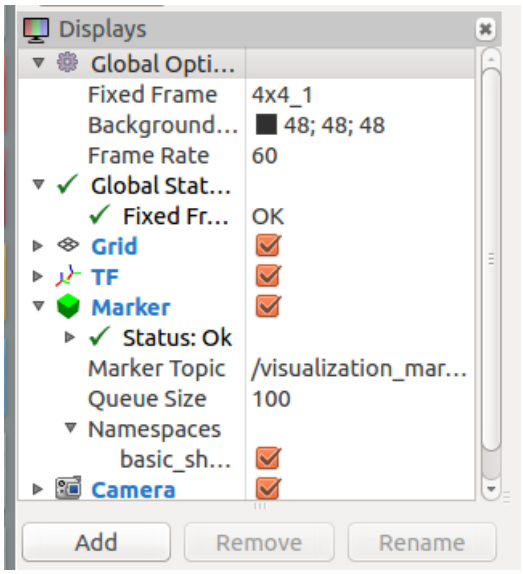


Figura 20. Ejemplo de interfaz de *rviz*. [Fuente: http://www.iheartrobotics.com/2011/09/rviz-and-ros-running-on-osx.html]

5.5.2 Aplicación en el proyecto

Rviz, a diferencia de los otros paquetes, será el único que no tenga un papel activo en el proyecto. Servirá para que los resultados conforme se vayan obteniendo puedan ser verificados de un modo más intuitivo que simplemente el ver valores en el terminal. En el fichero que se ha ido completando, *rviz_launcher.launch*, será el primer nodo que se ejecute.

El archivo de configuración que se usará como parámetro, en este caso el nombre del fichero será *rviz_beacon_detection.rviz,* se puede modificar en función de las preferencias. *Rviz* detecta una serie de *topics* particulares, entre los que se encuentran marcadores de visualización o transformadas. Por tanto, el entorno permite seleccionar de todos esos *topics* que se encuentren publicados, cuáles son los que se quieren reproducir, y guardar dichas prestaciones para ser ejecutadas del mismo modo en cualquier momento. En este caso se van a destacar, a través de la siguiente imagen, cuáles son los elementos más remarcables que se encuentran en este archivo de configuración.



Como se puede ver en la figura 21 en primer se ha seleccionado como eje fijo de coordenadas el marcador (4x4_1), se ha añadido la rejilla, los ejes de coordenadas correspondientes tanto al marcador como a la cámara se encuentran seleccionados, se ha incluido el marcador que publica *ar_pose* y por último se ha incluido la recepción de imágenes por cámara.

Figura 21. Ventana de *displays* en *rviz.*

Por último se puede observar en la siguiente imagen como se comunican todos los nodos cuando el fichero al completo es ejecutado.

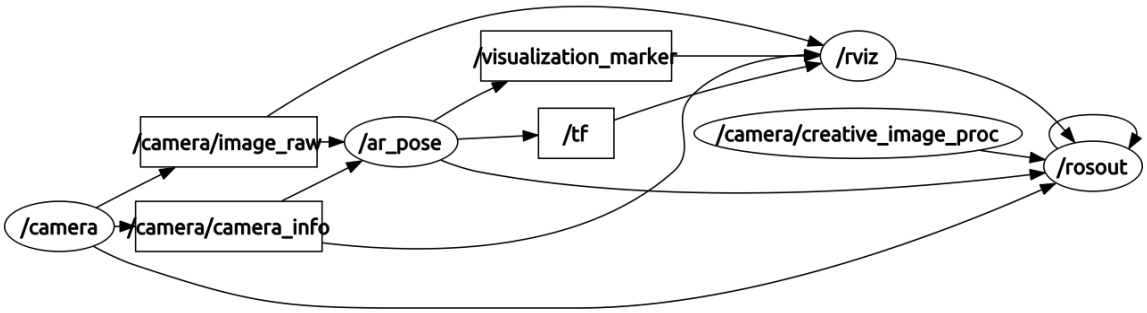


Figura 22. Interacción entre *ar_pose, gscam* y *rviz* en la Gráfica de Computación.

En la figura 22 se muestra como *rviz* se ha subscrito a todos y cada uno de los *topics* que publican los paquetes que se han descrito: se subscribe al *topic* de la cámara */image_raw*, y a dos *topics* que publica *ar_pose*, la transformada (*/tf*) y el marcador (*/visualization_marker*).

De este modo se puede recapitular como se ha ido completando el fichero *rviz_launcher.launch*:

1. Se ha creado el nodo correspondiente a la cámara, especificando a través de que *topic* se quería publicar, y qué fichero de calibración se iba a usar.

2. Se ha creado el nodo que ejecuta *ar_pose*, especificando principalmente el archivo donde se van a encontrar todos los marcadores almacenados.

3. Se ejecuta *rviz*, especificando el fichero de configuración que se va a emplear.

A partir de aquí solo queda crear el paquete que se empleará para procesar la información que se recibirá a través de *ar_pose*, y que completará el sistema que se está diseñando para el trabajo.

5.5.3 Resultados

Una vez que se ejecuta el fichero *rviz_launcher.launch* al completo se observa cómo en primer lugar se inicializa *rviz* en pantalla. Dentro del interfaz se muestran por un lado las imágenes detectadas por la cámara, los distintos elementos que se incluirán en el entorno, y el correcto funcionamiento y enlace de los arboles de transformadas y su aparición en el entorno 3-D. La reacción de todos estos elementos cuando se introduce un marcador en el marco de la cámara se observa en las figura 23:

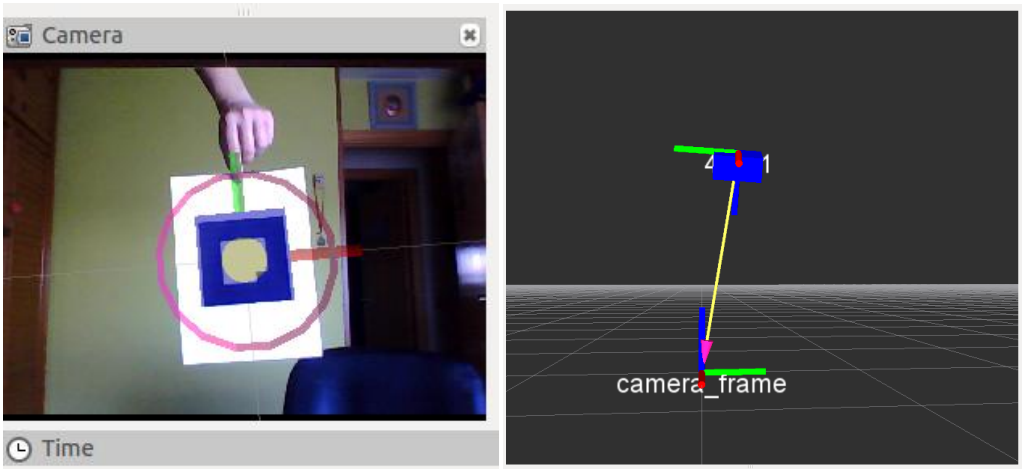


Figura 23. Izquierda) Cámara

Derecha) entorno 3-D en *rviz.*

Cuando se detecta el marcador, *ar_pose* envía información a través de varios *topics*, como se ha visto en el esquemático de la Grafica de Computación, acerca de la posición del marcador. Los dos *topics* que reconoce *rviz* son el marcador de visualización, que es el que superpone a la imagen recibida de la cámara (izquierda), y la información de la transformada que une a los dos ejes de coordenadas, que es lo que reproduce en el entorno virtual (derecha).

6. Diseño de la Aplicación

Una vez que ya se han especificado todas las herramientas que formarán inicialmente este sistema, se pasa a describir la aplicación que se creará originalmente para este trabajo.

A lo largo de este capítulo se irán describiendo los pasos que se han ido siguiendo para su creación. Comenzando por explicar cómo se crea un paquete correctamente en el entorno de *ROS*, qué herramientas se emplearán en el diseño del paquete, y una vez que estas bases se asienten, se pasará a describir explícitamente que pasos se han seguido a la hora de programar el contenido del paquete: qué clases se han creado, que funciones lo integran y en que bases matemáticas se asientan y cómo se integra este software en *ROS*, es decir, como publica y se subscribe. Se irá paso por paso hasta lograr construir el paquete en su estado más básico.

Pero antes de ello se ha de explicar cuál es la motivación del paquete y cuál es su función en el sistema que se ha de diseñar.

El ejecutable a diseñar tiene como objetivo es subscribirse al nodo del paquete *ar_pose, ar_multi*. De ese nodo, subscrito a su vez al ejecutable *gscam*, que controla la recepción de imágenes por cámara web, recibirá la información que *ar_pose* publica cada vez que detecta un marcador, es decir, la posición relativa del marcador con respecto a la cámara. Por otro lado, el paquete recibe información también desde un fichero *XML* en el que se encuentra toda la información respectiva a estos marcadores, que asocia su id con la posición absoluta que ocupan y su orientación con respecto a un punto denominado “Mundo”. Con estas dos fuentes de información el nodo recibe, cuando se detecta un marcador, dos transformadas: la que va desde la cámara al marcador y la que va desde el mundo al marcador. El objetivo final es, con esas dos transformadas, construir un árbol que logre referenciar la información de la cámara directamente al “mundo”, lo que significa obtener la transformada del “Mundo” hacia la cámara. Como se explicará más adelante, el nodo empleará las herramientas matemáticas necesarias para que esto sea posible, y una vez lo logre, publicará la información en un mensaje.

En la figura 24 se ha desarrollado esquemáticamente la lógica del paquete:

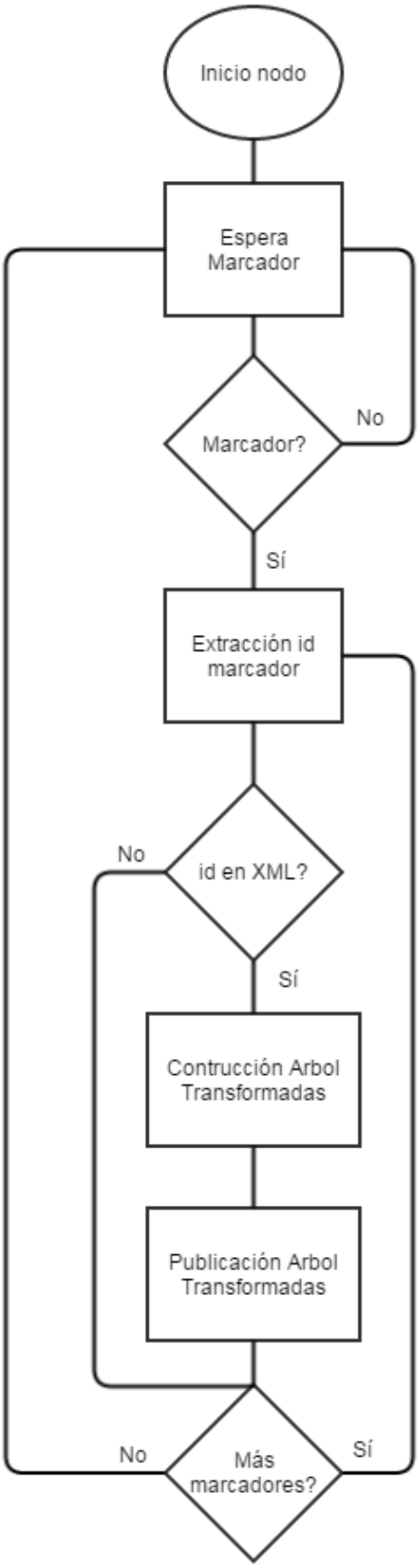


Figura 24. Diagrama de flujo del ejecutable al completo.

6.1 El Paquete beacon_detection

Una vez que se ha decidido cuál va a ser el nombre del paquete y se ha creado una carpeta en el *workspace*, hay que definir cuáles son los elementos principales que hacen de una simple carpeta un paquete para *ROS*. *ROS* trabaja con el sistema de paquetes *catkin*, para crear un paquete compatible con *catkin* se necesitan cumplir unos requerimientos [32]:

- El archivo *package.xml,* que posee meta-información sobre el paquete.
- Un archivo *CMakesLists.txt* que use *catkin*.
- Que no haya más de un paquete por carpeta

Los dos archivos que se mencionan son los que separan los conceptos de creación y construcción dentro de un paquete: el archivo *package.xml* confiere identidad al paquete dando toda la información relevante del mismo y sus dependencias, y el archivo *CMakeLists.txt* da toda la información necesaria para que la construcción se realice correctamente y acorde con el sistema *catkin*.

6.1.1 Contenido del Paquete

Para una mejor comprensión de los siguientes apartados lo primero que se ha de especificar es cuál es el contenido del paquete.

- ***package.xml***. Es el manifiesto del paquete y es uno de los primeros archivos que se crean al ejecutar el comando de *ROS catkin_create_pkg*.
- ***CMakeLists.txt.*** Contiene toda la información relativa a la construcción del paquete en ROS.
- ***include*.** En esta carpeta se almacenan todos los *header* del paquete, es decir, los *.h*, *.hpp*, etc. En este caso los ficheros almacenados son los que se muestran en la imagen.

Figura 25. Carpeta *include* en el paquete *beacon_detection.*

- ***src.*** En esta carpeta se almacenan todos los *source* del paquete, los archivos *.cpp*, que se pueden emplear como bibliotecas y ejecutables en ROS. En la imagen se puede ver el contenido de esta carpeta.

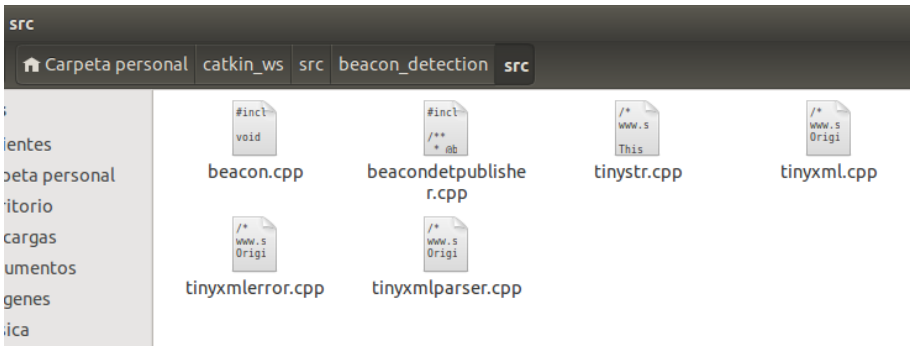


Figura 26. Carpeta *src* en el paquete *beacon_detection.*

- ***files*.** Aquí se almacenarán todos los ficheros *XML* con la información de la posición y orientación de los marcadores con respecto a un sistema de referencia.

- ***launch*.** Aquí se almacenarán todos los ficheros *.launch* que se ejecutan empleando el comando *roslaunch* de *ROS*. El archivo *rviz_launcher.launch* desarrollado en el capítulo anterior es un ejemplo del contenido de esta carpeta.

Este fragmento es un índice final del contenido del paquete. A lo largo de este capítulo se profundizará en las carpetas *include* y *src*, indicando cómo y por qué se van añadiendo los archivos que las componen.

6.2 Programación

Una vez que las herramientas que se van a emplear para programar están explicadas y el paquete se encuentra totalmente construido, se pasa a la parte más importante del diseño de la aplicación: la programación. Se va a ir abordando a lo largo de la sección cada parte del código, según el orden en el que se ha ido creando y al concepto al que está asociado.

6.2.1 Lectura de fichero *XML*

El primer paso con el que se comienza el código es leyendo un fichero en el que se van a encontrar las coordenadas de la baliza y su orientación con respecto a un eje de coordenadas origen al que se llamará coloquialmente “Mundo” o “World”. Para ello se emplea la herramienta descrita anteriormente, *TinyXml*. Entre las múltiples ventajas que se han destacado, se encuentra su fácil implementación al código, siendo únicamente necesario el incluir cada uno de los ficheros dentro del propio paquete. Una vez que este paso se ha cumplido, falta determinar cómo se incluye en el código original.

6.2.1.1 Archivo *Beacons.xml*

Lo primero que hay que diseñar es el fichero en el que va a ir incluida la información y cómo va a ir incluida. Lo que está claro es que la posición se va indicar en coordenadas cartesianas, pero la cuestión es cómo encapsular la información con respecto a la orientación.

En este punto hay múltiples opciones de representación de orientaciones en el espacio, pero se pretende que la interacción humano-máquina sea lo más intuitiva posible. Por esto, se van a emplear los ángulos de navegación *Roll*, *Pitch* y *Yaw* para representar los giros. Siendo el criterio en este caso de que *Roll* es el giro alrededor del eje *x*, *Pitch* alrededor de *y,* y *Yaw* alrededor de *z*.

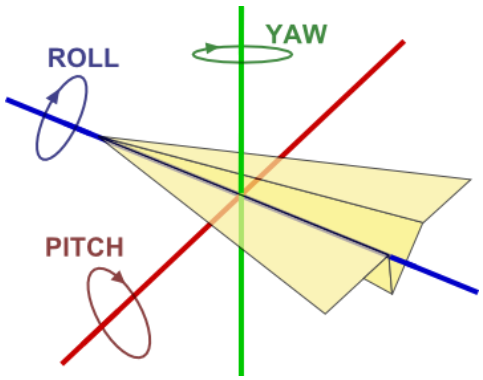


Figura 27. Ángulos de navegación.

De este modo, el aspecto del fichero será como se muestra en la siguiente imagen:

```
<?xml version="1.0" ?>
<root>
    <Beacon1 id="0" x="5" y="-1.25" z="1.2" roll="1.57" pitch="0" yaw="3.14"/>
    <Beacon2 id="1" x="9.75" y="1.10" z="1.2" roll="1.57" pitch="0" yaw="0"/>
    <Beacon3 id="2" x="1" y="2" z="2" roll="1.57" pitch="0" yaw="0"/>
</root>
```

Figura 28. Fragmento del fichero *beacons.xml.*

Como se puede ver solo hay una capa de herencia, el objeto principal posee sus coordenadas y su orientación, expresada en los ángulos de navegación. Por último, cada *Beacon* lleva asociado un número, un elemento clave a la hora de extraer la información selectivamente.

6.2.1.2 Clase *Beacon*

Una vez que se ha decidido cómo organizar el fichero *XML*, el siguiente paso es cómo almacenar la información que se pretende extraer. Para esto se va a crear la clase *Beacon*, que no es más que una estructura dispuesta para este fin. Antes de comenzar a describir su contenido se han de destacar dos puntos. En primer lugar, la clase estará formada por un *header* y un *source*, este último, en función si contendrá o no la función *main*, se incluirá en el fichero *CMakeLists.txt* como ejecutable o como biblioteca. En segundo lugar, al ser el primer eslabón del código del paquete, el fichero *header* será el archivo en el que se incluirán todas las bibliotecas y hacia el que se referenciarán el resto de archivos conforme se vayan creando.

```
double _x;
double _y;
double _z;
double _roll;
double _pitch;
double _yaw;
```

Como se puede ver en la imagen, la clase llevará incluida toda la información que procede del fichero a excepción del *id*, ya que este último se usa únicamente para encontrar la baliza correcta dentro del *XML* y además solo se pretende publicar una baliza cada vez, por tanto no habrá necesidad de identificar la baliza dentro del código.

El resto del código incluirá todos los métodos que acompañan a una clase tan básica, es decir, métodos que permitan extraer e introducir la información, y los distintos operadores que permitirán realizar operaciones entre las balizas en caso de ser necesario. Uno de los operadores que se destacan en este caso es el operador *OS* que permite imprimir toda la información de la baliza a través de dicho *stream* y que será de una gran practicidad a la hora de comprobar que la información se extrae correctamente del fichero.

```
friend std::ostream &operator <<(std::ostream &OS, const Beacon &B)
{
    OS<<"x="<<B._x<<" y="<<B._y<<" z="<<B._z<<" roll="<<B._roll<<" pitch="<<B._pitch<<" yaw="<<B._yaw;
    return OS;
}
```

6.2.1.3 Función *readXml*

Este punto es el más relevante con respecto a la lectura de fichero, ya que es el punto en el que interaccionan *TinyXml* y el paquete. Lo primero que se ha de indicar acerca de la función es que posee un *input*, que será el *id* que se pretende buscar en la función, y un output, un objeto de la recién creada clase *Beacon* con la información contenida en el fichero correspondiente a ese *id*.

La función se divide internamente en varios puntos. En primer lugar, se ha de verificar que hay un fichero y un elemento raíz o *root*.

```
TiXmlDocument doc;
int id;
double x, y, z, roll, pitch, yaw;
Beacon B;

if(!doc.LoadFile(FILE_PATH))
{
    cerr << doc.ErrorDesc() << endl;
    _publish = false;
    return B;
}
TiXmlElement* root = doc.FirstChildElement();
if(root == NULL)
{
    cerr << "Failed to load file: No root element."
         << endl;
    doc.Clear();
    _publish = false;
    return B;
}
```

La variable booleana que aparece como *_publish* es una variable global que se emplea para que en caso de que no se encuentre fichero alguno, el código tenga un modo de controlar y comunicar que no se publique la información.

A continuación la función va sacando elemento por elemento la información del fichero XML e introduciéndolo en un objeto de la clase *Beacon*, el *output* de la función.

En la figura XX se muestra un flujograma completo de la función:

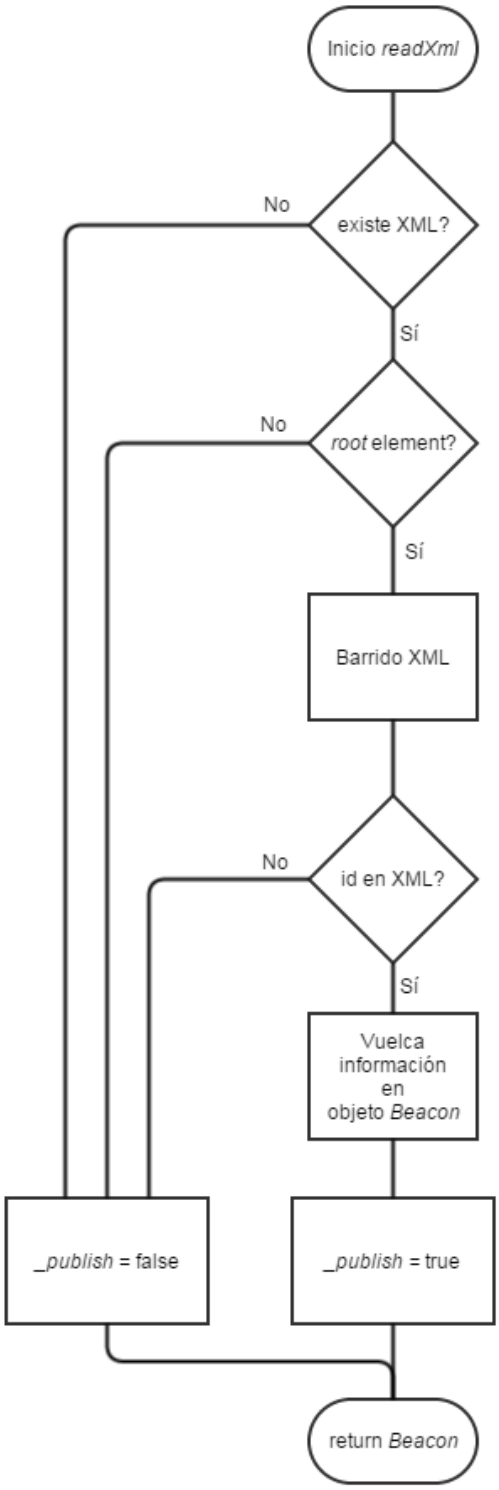


Figura 29. Flujograma función *readXml.*

En un primer momento se barre todo el fichero en la búsqueda del *id* y en caso de encontrarse, es cuando se extrae toda la información.

De nuevo, en caso de que no se encuentre en el fichero la baliza deseada, la función indicará a través de la variable *_publish* que la información no sea publicada.

6.2.2 Inicialización del nodo

En este momento el código es capaz de recibir información que recibe de un fichero *XML*, pero aun no se encuentra dentro del sistema de comunicaciones de *ROS* como tal. Para que esto sea posible se han de realizar una serie de modificaciones sobre la función de inicialización de un código por excelencia, el *main.*

6.2.2.1 beacon_detection_node

```
int main (int argc, char **argv)
{
    ros::init (argc, argv, "beacon_detection_node");
    ros::NodeHandle n;

    return 0;
}
```

De esta función se han de destacar principalmente dos detalles. En primer lugar, el método *init* es lo que hace que el código pueda ejecutarse bajo *ROS*, es lo que convierte primordialmente un código ejecutable en un nodo. El segundo objeto que se inicializa es lo que se denomina *NodeHandle* y se trata del objeto que permite que el nodo se comunique dentro del entorno de *ROS*. El archivo *.cpp* en el que se incluya la inicialización, es decir, el método *main*, será el que se indique en el *CMakeLists.txt* como ejecutable, en vez de como biblioteca.

6.2.3 Recepción de la Información

Una vez que es nodo se encuentra correctamente inicializado, el siguiente paso es conseguir que este se subscriba a *ar_pose*. Para ello se han de hacer modificaciones tanto en el *main*, como crear la función *callback* que será la que reaccione en caso de que el nodo logre subscribirse.

6.2.3.1 Clase *ros::Subscriber*

En un primer lugar se ha de centrar la atención sobre la función *main* de nuevo. Se le van a realizar las siguientes modificaciones, tal y como se aprecia en la imagen.

```
int main (int argc, char **argv)
{
    ros::init (argc, argv, "beacon_detection_node");
    ros::NodeHandle _n;

    ros::Subscriber _sub = _n.subscribe("ar_pose_marker", 1000, callback);

    ros::spin ();
    return 0;
}
```

Para subscribirse se ha de crear un objeto de la clase *Subscriber,* que se inicializa a través del *NodeHandle* y su método *subscribe*. La función *subscribe* tiene como *inputs* el *topic* al que se subscribe, la cantidad de mensajes que como máximo se van a acumular y, por último y más importante, la función a la que va a acudir cuando reciba información de dicho *topic*. En este caso y para que sea más intuitivo el nombre de dicha función será *callback*.

Por otro lado se ha incluido una función con el nombre de *spin*. La función que cumple *spin* es la de básicamente esperar hasta que se reciba información del *topic* al que se va a subscribir el nodo; es un *loop* infinito que ejecuta la función indicada en la inicialización del *Subscriber* cada vez que esto ocurra.

6.2.3.2 Función *callback*

Por último, para que el nodo sea capaz de subscribirse queda diseñar la función *callback*. Se trata de un método que representa una respuesta del nodo a la recepción de información desde un *topic* al que se ha subscrito. Es un punto que sufre varias modificaciones a lo largo del trabajo, pero en esta parte de lo que se tiene que ocupar es únicamente de recibir la información correctamente y ser capaz de extraerla. Para ello, en primer lugar se ha de incluir como biblioteca el *header* ***"ar_pose/ARMarkers.h"*** que incluye los marcadores como mensajes, para que estos puedan ser utilizados.

La función *callback* tendrá el siguiente aspecto:

```
void callback(const ar_pose::ARMarkers::ConstPtr& msg)
{
    ar_pose::ARMarker ar_pose_marker;

    double roll, pitch, yaw;

    for(int i=0;i < msg->markers.size();i++){
        ar_pose_marker = msg->markers.at(i);

        tf::Quaternion q(ar_pose_marker.pose.pose.orientation.x,ar_pose_marker.pose.pose.orientation.y
                         ,ar_pose_marker.pose.pose.orientation.z,ar_pose_marker.pose.pose.orientation.w);
        tf::Matrix3x3 m(q);

        // The data is displayed in Terminal
        m.getRPY(roll, pitch, yaw);
        std::cout<<"id " << ar_pose_marker.id<<std::endl;
        std::cout<<"x " << ar_pose_marker.pose.pose.position.x<<std::endl;
        std::cout<<"y " << ar_pose_marker.pose.pose.position.y<<std::endl;
        std::cout<<"z " << ar_pose_marker.pose.pose.position.z<<std::endl;
        std::cout<<"Roll "<<(roll*180)/PI<<endl;
        std::cout<<"Pitch "<<(pitch*180)/PI<<endl;
        std::cout<<"Yaw "<<(yaw*180)/PI<<endl;

    }

}
```

La función se divide en varios puntos relevantes que se han de explicar por separado. En primer lugar el input de la función es un puntero que contiene una *array* de marcadores. Las variables que se crean son, por un lado tres *double* que contendrán la información de los ángulos de navegación que se extraigan de los marcadores de *ar_pose,* y por otro, un objeto de la clase *ARMarker* para extraer la información de la *array* de marcadores que se encuentra dentro del puntero antes mencionado. Para extraer la información en caso de que haya simultáneamente varios marcadores se inicia un bucle que los analice por separado.

```
void callback(const ar_pose::ARMarkers::ConstPtr& msg)
{
    ar_pose::ARMarker ar_pose_marker;

    double roll, pitch, yaw;

    for(int i=0;i < msg->markers.size();i++){
        ar_pose_marker = msg->markers.at(i);
```

A continuación se hace uso de la biblioteca antes mencionada *tf* para hacer la conversión de cuaterno a ángulos de navegación. Para ello se crea un objeto de matriz 3x3 en la que se introduce el cuaterno con la información del marcador, luego se emplea el método *getRPY* propio de la clase *tf::Matrix3x3* con los *doubles* que se crearon antes como inputs. El método *getRPY* introduce los valores de estos ángulos dentro de dichos inputs empleando referencias.

```
tf::Quaternion q(ar_pose_marker.pose.pose.orientation.x,ar_pose_marker.pose.pose.orientation.y
                 ,ar_pose_marker.pose.pose.orientation.z,ar_pose_marker.pose.pose.orientation.w);
tf::Matrix3x3 m(q);
m.getRPY(roll, pitch, yaw);
```

Por último solo queda mostrar dicha información por pantalla. *ROS* es perfectamente compatible con los *streams* de *C++,* por lo que únicamente hace falta emplear el método *cout*.

6.2.3.3 Resultados

Una vez que se ha construido el código sólo queda ejecutar *ar_pose* y posteriormente el nodo. Para ello se ejecuta el comando *rosrun* con el nombre del paquete y el nombre del nodo.

```
rosrun beacon_detection beacon_detection_node
```

Y una vez que se ejecute se podrá comprobar si se subscribe *ar_pose* ejecutando el nodo *rqt_graph*.

Figura 30. Subscripción de *beacon_detection* a *ar_pose.*

Y si todo el sistema está subscrito adecuadamente, es decir *ar_pose* a la cámara, y *beacon_detection* a *ar_pose*, y se muestra en la cámara un marcador, por el terminal en el que se ha ejecutado el nodo de *beacon_detection* se comenzarán a emitir las coordenadas y la orientación del marcador, indicando que todo el código funciona correctamente.

6.2.4 Gestión de la información

Hasta ahora se ha logrado que el nodo se inicialice y reciba información de las dos fuentes principales. Lo que se tiene que conseguir a continuación es gestionar la información de tal modo que se pueda cumplir el objetivo de situar la cámara con respecto a un sistema de coordenadas absoluto.

6.2.4.1 Fundamento teórico

En primer lugar se ha de definir que es la información a gestionar. Se tienen tres sistemas de coordenadas:

- **"Mundo" o "World".** El sistema de coordenadas origen hacia el que se quiere referenciar toda la información del entorno.
- **"Marcador".** Un marcador fijo que se pretende situar en un determinado punto del entorno.
- **"Cámara".** Representa el robot móvil.

Y la información que se recibe es la siguiente:

- **Transformada de "Mundo" a "Marcador" o $T^{W}{}_{M}$.** La información que está contenida en el fichero *XML*.
- **Transformada de "Cámara" a "Marcador" o $T^{C}{}_{M}$.** Ésta es la información que se obtiene de *ar_pose*.

Ahora lo que se ha de determinar es con qué orientación y cómo están situados en su estado original estos elementos. La orientación de la primera transformada ("Mundo" a "Marcador") será la que el usuario desee, pero en el caso de *ar_pose*, se ha de verificar qué orientación relativa tiene el marcador con respecto a la cámara. Una vez que esto se compruebe, se realizarán sobre esta transformada las operaciones necesarias para orientarla como se crea conveniente. En la imagen a continuación se puede ver cómo está orientado el marcador con respecto a la cámara.

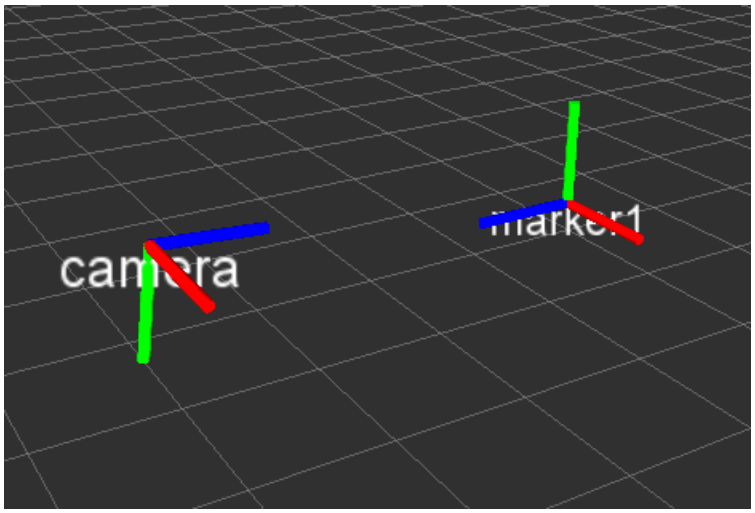


Figura 31. Relación de los ejes Marcador y Camara.

Como se puede ver el marcador se encuentra girado 180 grados en el eje *x* (*Roll*) con respecto a la cámara. Esto se ha de modificar de dos maneras, en un primer lugar la información que devuelve *ar_pose* establece que el eje de coordenadas fijo es la cámara y el móvil, el marcador. Si se quiere establecer lo opuesto, lo que se ha de hacer es invertir esta transformada.

Se tiene una transformada representada por una matriz homogénea del siguiente modo:

$$\boldsymbol{T} = \begin{bmatrix} \boldsymbol{R}_{3x3} & \boldsymbol{p}_{3x3} \\ \boldsymbol{f}_{1x3} & \boldsymbol{w}_{1x1} \end{bmatrix} = \begin{bmatrix} Rotación & Traslación \\ Perspectiva & Escalado \end{bmatrix} \quad (1)$$

Siendo la perspectiva y el escalado 0 y 1 respectivamente. Teniendo en cuenta que la matriz de rotación $\mathbf{R_{3x3}}$ en (1) es ortonormal, es decir que su inversa es su matriz traspuesta, la inversa de la matriz al completo será:

$$T = \begin{bmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}; \; T^{-1} = \begin{bmatrix} n_x & n_y & n_z & -\boldsymbol{n}^T\boldsymbol{p} \\ o_x & o_y & o_z & -\boldsymbol{o}^T\boldsymbol{p} \\ a_x & a_y & a_z & -\boldsymbol{a}^T\boldsymbol{p} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2)$$

En este punto ya se ha referenciado correctamente la cámara al marcador, estableciendo que la cámara es móvil y el marcador fijo; con (2) se ha pasado de ${T^C}_M$ a ${T^M}_C$. El siguiente paso es rotar (2) para que el eje que represente la cabeza del sistema de coordenadas sea, en vez del eje *z*, el eje *x*, de modo que el movimiento de la cámara sea más intuitivo.

Está operación se aborda matemáticamente de la siguiente manera. Para establecer rotaciones consecutivas, se ha de multiplicar la matriz de rotación deseada, en este caso la que está contenida en la matriz homogénea ${T^M}_C$**,** por las rotaciones que deseen hacerse en el orden que se desee, en este caso para que el sistema de coordenadas tenga como eje frontal el eje *x* se ha de rotar primero 90 grados en *Roll* y posteriormente -90 en *Pitch*. Esto queda representado matemáticamente en (3).

$${R^M}_{C_{final}} = {R^M}_C \times R(90^o x) \times R(-90^o y) \quad (3)$$

Una vez que la transformada ya está referenciada y rotada adecuadamente, queda referenciarla con respecto al mundo en vez de al marcador. Para ello se empleará la siguiente fórmula.

$${T^W}_C = {T^W}_M \times {T^M}_C \quad (4)$$

En (4) ${T^W}_M$ es la información que proviene del fichero *XML* y ${T^M}_C$ es la transformada que se ha calculado que va del marcador a la cámara.

6.2.4.2 Creación del Árbol de Transformadas en el Código

Toda esta información que se ha calculado teóricamente se ha de pasar a código de algún modo. Emplear matrices homogéneas puede ser una tarea compleja sobretodo en un lenguaje que no está preparado para trabajar específicamente con matrices, en contraposición con software ideado específicamente para ello (por ejemplo *Matlab*). Pero para realizar este trabajo, *ROS* posee la biblioteca citada al principio del capítulo, *tf*, que posee todas las clases y funciones necesarias para trabajar con estos elementos.

En un primer lugar, para emplear la biblioteca, ésta se ha de incluir en el fichero en el que se vayan a incluir todas las bibliotecas empleando la siguiente línea:

#include <tf/transform_broadcaster.h>

Incluyendo esto, ya que es la clase que contiene al resto, se están incluyendo todas las herramientas, incluido el publicar directamente las transformadas en caso de que se desee emplear el paquete *rviz* para visualizar resultados.

Una vez que se ha incluido esta línea, solo queda emplear las operaciones necesarias. Para ello se van a ir separando las tareas y su aparición en la función *callback*, que es donde se va a hacer todo este procesamiento.

1. **Obtención de la transformada $\boldsymbol{{T^W}_M}$.** Al obtener el primer marcador, en caso de que hubiera varios, se introduce se introduce su id en la función *readXml* y se obtiene un objeto de la clase *Beacon* con el mismo id que se ha introducido.

```
Beacon B;

for(int i=0;i < msg->markers.size();i++)
{
    ar_pose_marker = msg->markers.at(i);

    B = readXml(ar_pose_marker.id);
```

2. **Obtención de la transformada $\boldsymbol{{T^M}_C}$.** A través de la información que se obtiene del marcador se crean una matriz de rotación y un vector de traslación, con ellos se crea la matriz homogénea. Esta matriz, siguiendo el proceso, la ecuación (2), se invierte empleando el método *inverse*. En este punto también se realiza el mismo proceso con la *Beacon* correspondiente a la transformada ${T^W}_M$, creando el vector de traslación y la matriz de rotación, y con ellos la matriz homogénea *m_file*.

```
tf::Quaternion q(ar_pose_marker.pose.pose.orientation.x,ar_pose_marker.pose.pose.orientation.y,
                 ar_pose_marker.pose.pose.orientation.z,ar_pose_marker.pose.pose.orientation.w);
tf::Matrix3x3 m(q);
tf::Matrix3x3 m_file;
m_file.setRPY(B.getRoll(),B.getPitch(),B.getYaw());
tf::Vector3 v(ar_pose_marker.pose.pose.position.x,ar_pose_marker.pose.pose.position.y,
              ar_pose_marker.pose.pose.position.z);
tf::Vector3 v_file(B.getx(), B.gety(), B.getz());
tf::Transform t(m, v);
tf::Transform t_file(m_file, v_file);
tf::Transform inv = t.inverse();
```

3. **Rotación de la transformada $T^{M}{}_{C}$.** Se rota esta matriz 90 grados en *roll* y -90 en *pitch*, para que el eje *x* sea el frontal. Para ello se crea una sola matriz con los dos giros, ya que son teóricamente consecutivos. Esta parte está ilustrada matemáticamente en la ecuación (3).

```
tf::Matrix3x3 m_aux;
m_aux.setRPY(PI/2,-PI/2,0);
inv.setBasis(inv.getBasis() * m_aux);
```

4. **Obtención de la transformada $T^{W}{}_{C}$.** Se realiza el producto de matrices homogéneas siguiendo la fórmula explicada anteriormente en (4).

```
tf::Transform T = t_file * inv;
```

6.2.4.3 Resultados

Los resultados que se esperan obtener son la verificación de que estos cálculos son correctos y que están plasmados correctamente en el código. En primer lugar se va a emplear el objeto *Transform_broadcaster* para emitir las transformadas. Mediante este objeto se pueden publicar transformadas con sello*, Stamped_transform* (son las transformadas al uso pero además contienen información del nombre de los ejes y del momento en el que se crean), sin necesidad de incluirlas en un mensaje al uso, siendo además un formato que recibe *rviz*. [Nota: *Transform_broadcaster* es únicamente una herramienta que se empleara por motivos de verificación, no es esencial en el proceso de publicación.]

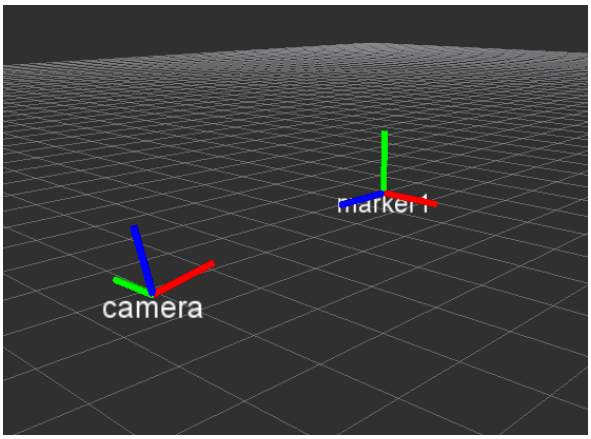


Figura 32. Resultados en *rviz*.

A parte de esto, aunque será menos intuitivo se va a imprimir por terminal también la información. La figura 33 muestra estos resultados:

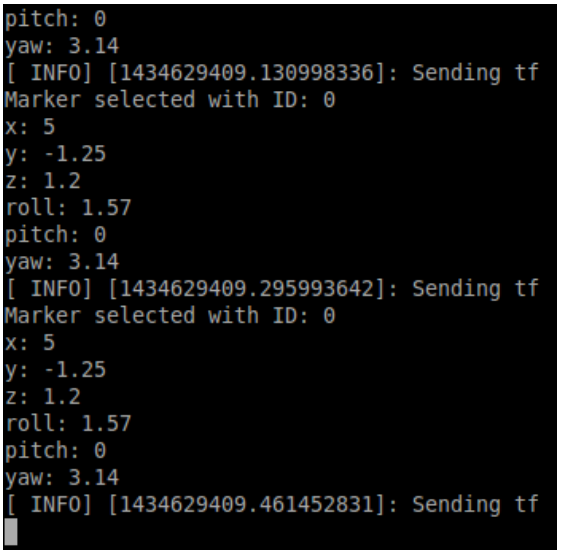


Figura 33. Resultados mostrados por terminal.

Y como se puede ver en las imágenes, la información que se obtiene a través de *rviz* verifica que el proceso se ha realizado correctamente.

6.2.5 Publicación de la información

Una vez que la aplicación se subscribe correctamente y es capaz de procesar correctamente la información que le llega tanto por subscripción como por el fichero *XML*, el siguiente y último paso es el de publicar la información procesada a través de un *topic*.

6.2.5.1 Planteamiento y problemas

La cuestión determinante en este caso es que se pretende publicar en el *callback*. Aparentemente esto puede no parecer un problema, la cuestión es que el *Publisher*, al igual que el *Subscriber*, ha de ser inicializado en el *main* empleando un objeto de la clase *NodeHandle.*

```
pub = n.advertise<geometry_msgs::TransformStamped>("worldToCamera", 1);
```

En la función *advertise* se emplea como *template* el mensaje a publicar, y como *inputs* el nombre del *topic* sobre el que se publica y la cantidad de mensajes que se quieren acumular en este.

Del modo en que funcionan las comunicaciones en *ROS*, el método *callback* es hermético, lo que no permite que tenga *inputs* al margen del mensaje a recibir y tampoco *outputs*. Por lo que si se pretende publicar la información que se reciba, no hay manera de hacerlo ya que, por un lado no se puede sacar la información a través de un *output*, y por el otro, si el *Publisher* se inicializa en el *main*, el *callback* no tiene modo de saber de su existencia. Para resolver este problema se abren dos caminos a seguir.

Una de las opciones son las variables globales. El problema es que aparentemente no hay un lugar donde almacenar de modo absoluto esa información. Una serie de variables globales parece que resolverían dicha dificultad. Esta opción tiene dos desventajas esencialmente. La primera es que si lo que se pretende es crear un Publisher global no es posible al uso, ya que depende de la previa inicialización de un *NodeHandle*, que a su vez depende de que se inicialice el nodo, por lo que si se quisiese emplear un Publisher global se necesitaría que fuera un puntero. En segundo lugar, y casi apoyando el primer argumento, programar empleando variables globales es un método considerado como "sucio" ya que como contenedores globales son más apropiados tanto las estructuras como las clases.

De este modo esta segunda opción, crear una nueva clase, parece la opción más adecuada a seguir. Para ello se han de realizar una serie de modificaciones en todo el código, para que éste se adapte a esta nueva clase.

6.2.5.2 Clase *BeaconDetPublisher*

En primer lugar se han de crear los archivos que contendrán la clase, el *.cpp* y el *.h*. Una vez que están creados y correctamente incluidos en el archivo *CMakeLists.txt* como bibliotecas o como el ejecutable (en caso de que el *main* se incluya en su *.cpp*), se procede a ir creando la clase. Los elementos que inicialmente tiene que incluir la clase serán los elementos que se ha indicado que dependen unos de otros, es decir, el *NodeHandle*, el *Publisher* y el *Subscriber*. Además incluirá la variable global *_publish* que en la función *readXml* controlaba las excepciones y un *Transform_broadcaster* para enviar los resultados al visualizador *rviz*.

En cuanto a los métodos que se incluyen dentro de la clase, se listan prácticamente todos los que se han creado hasta ahora excepto el *main*, es decir, *readXml* y *callback*.

Cuando se inicialice el objeto, el constructor tendrá como input el *NodeHandle* del *main*, que servirá para crear el propio *NodeHandle* del objeto, tras esto, empleará el *NodeHandle* para inicializar tanto el *Publisher* como el *Subscriber.* Ésta es una muestra del *main*.

```
int main (int argc, char **argv)
{
    ros::init (argc, argv, "beacon_detection_node");
    ros::NodeHandle n;
    BeaconDetPublisher beac_detection_publisher (n);
    ros::spin ();
    return 0;
}
```

Y éste es el constructor de la clase.

```
BeaconDetPublisher::BeaconDetPublisher(ros::NodeHandle &n):_n (n)
{
    //Topic you want to publish
    _pub = _n.advertise<geometry_msgs::TransformStamped>("worldToCamera", 1);

    ROS_INFO ("Subscribing to ar_pose_marker");
    _sub = _n.subscribe("ar_pose_marker", 1000, &BeaconDetPublisher::callback, this);
}
```

Como se puede ver en las funciones, la inicialización del *Subscriber* tiene una particularidad, y que se debe a que el *callback* se incluye en la propia clase: cuando se incluye el *callback* como *input* se emplea una referencia y un *input* más, un puntero apuntándose a sí mismo, *this*. El resto de los elementos son exactamente los mismos: el *topic* y el número máximo de mensajes que acumula.

A parte de tener que incluirse en la clase, las funciones apenas han de cambiarse. La única que se ha de modificar será *callback,* a la que se le han de añadir las siguientes líneas para que el nodo publique.

```
if (_publish = true)
{
        ROS_INFO ("Sending tf");

        // Here the transform is broadcasted
        tf::StampedTransform worldToCam (T, ros::Time::now(), "world", "camera");
        _tb.sendTransform(worldToCam);

        // The information is published
        tf::transformStampedTFToMsg(worldToCam, TS);
        _pub.publish(TS);

}
```

En primer lugar, se ha de pasar la publicación por el *bool _publish*, ya que en caso que surja algún problema en el *XML* o se quiera controlar globalmente la publicación, es el mejor modo de comunicarlo. Después de esto, se envía la información empleando un *Transform_broadcaster* para que los resultados se puedan visualizar en *rviz*. Por último, se emplea el método de la biblioteca *tf transformStampedTFToMsg* para convertir éste objeto propio de *tf* en un mensaje con los mismos campos y así poder publicarlo empleando el *Publisher* de la clase.

6.2.5.3 Resultados

Los resultados de la correcta publicación se pueden observar ejecutando todo el entramado de nodos y este último, y si todo se ha hecho correctamente, el nodo será capaz de subscribirse y de publicar la información en caso de visualizar la cámara un marcador. En la figura 34 se muestra a través de *rqt_graph* la correcta subscripción y publicación.

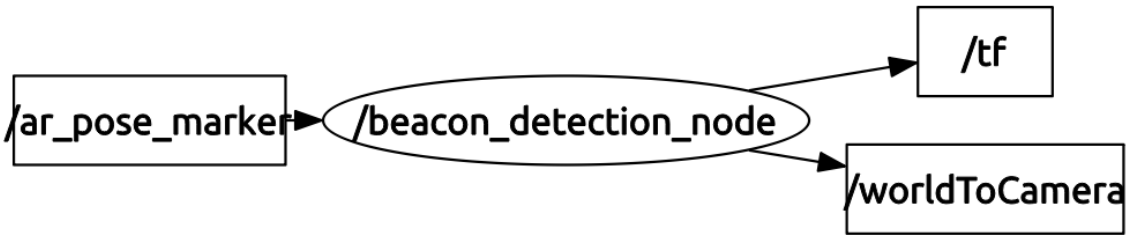


Figura 34. *Beacon_detection* publicando.

6.2.6 Ampliaciones

Hasta este último punto, el nodo que se ha diseñado es capaz de subscribirse a *ar_pose*, recibir información de un fichero *XML*, procesar estas dos últimas informaciones y publicar el resultado en *ROS*. Aún así, se han realizado algunos añadidos al código para hacerlo más fiable y darle más robustez.

6.2.6.1 Selección de Marcador

Según el código que se ha desarrollado hasta ahora, cuando se recibe un conjunto de marcadores, el *callback* los irá cogiendo, procesando y enviando uno por uno. El problema en este caso es que no se filtra la información que se va a enviar y por tanto se da la misma prioridad a todos los marcadores. El objetivo es seleccionar un criterio por el cual, en caso de estar visualizando varios marcadores al mismo tiempo, se seleccione el que más lo cumpla El criterio que se emplea en este caso es que se selecciona el que este situado con menor ángulo de inclinación e la cámara, es decir, el más perpendicular a ella.

Para ello se diseña la siguiente función.

```
int BeaconDetPublisher::getLesserAngle(double *angles, int size)
{
    double temp = PI;
    int j = 0;

    for(int i=0;i<size;i++)
    {
        if(abs(angles[i]) < abs(temp))
        {
            temp=angles[i];
            j = i;
        }
    }
    return j;
}
```

Esta función posee como inputs, un *array* de *doubles* y su tamaño. La idea es almacenar todos los *doubles* de los marcadores que se reciben, y que la función los compare todos y que indique en qué posición se encuentra el marcador con menor ángulo.

Todos estos cambios se han de realizar en el *callback*. Lo primero es que todo el procesamiento y publicación se extrae del bucle, quedando de este modo.

```
double *pitchs = new double[msg->markers.size()];
double null;

for(int i=0;i < msg->markers.size();i++)
{
    ar_pose_marker = msg->markers.at(i);

    tf::Quaternion q(ar_pose_marker.pose.pose.orientation.x,
                     ar_pose_marker.pose.pose.orientation.y,
                     ar_pose_marker.pose.pose.orientation.z,
                     ar_pose_marker.pose.pose.orientation.w);
    tf::Matrix3x3 m(q);

    m.getRPY(null, pitchs[i], null);
}
```

En el bucle se extraen todos los ángulos que contienen la inclinación lateral del marcador con respecto a la cámara, que en este caso según la información de *ar_pose* es *Pitch*. Posteriormente se asigna como marcador resultante es que se obtenga de la función anterior, añadiéndose la siguiente línea tras el bucle.

```
ar_pose_marker = msg->markers.at(getLesserAngle(pitchs, msg->markers.size()));

cout<<"Marker selected with ID: "<<ar_pose_marker.id<<endl;
```

A partir de aquí se somete al mismo procesamiento al marcador y se publica posteriormente el resultado. En la figura 35 se muestra el flujograma de cómo influye esta ampliación en el proceso completo.

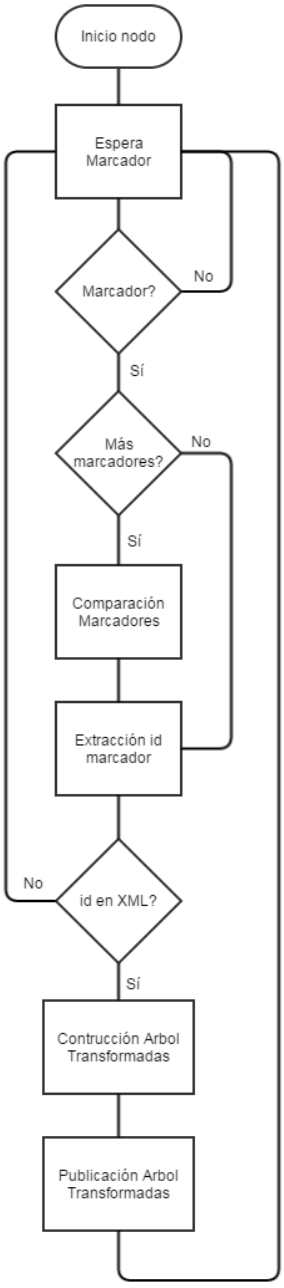


Figura 35. Flujograma del ejecutable con Ampliación 1

6.2.6.2 Eliminación de *outliers*

Esta modificación surge como respuesta a un fenómeno que se observa a lo largo de algunas pruebas. En algunos casos, cuando se mantenía un marcador en la pantalla, entre la información parpadeaba un marcador en la distancia, dando un falso positivo.

Para solucionarlo, la solución aparentemente más simple es almacenar el último marcador obtenido y comparar el siguiente con este último. Por ello se crea como variable de la clase *BeaconDetPublisher* un marcador que se actualizará en caso de que el marcador no sufra un cambio muy brusco en sus valores de posición y orientación. Para establecer esta comparación se crea la función *compareMarkers* que se incluye en la clase *BeaconDetPublisher* que, tomando como *inputs* dos marcadores, verifica que no haya una gran variación entre uno y otro. Si la variación es aceptable, se devuelve un *boolean true*, en caso contrario un *false*.

Como se puede ver en el siguiente segmento de la función *compareMarkers*, en primer lugar se comprueban las distancias.

```
bool BeaconDetPublisher::compareMarkers(ar_pose::ARMarker marker1, ar_pose::ARMarker marker2)
{
    double roll1, roll2, pitch1, pitch2, yaw1, yaw2;

    if(abs(marker2.pose.pose.position.x - marker1.pose.pose.position.x) > 2)
        return false;
    if(abs(marker2.pose.pose.position.y - marker1.pose.pose.position.y) > 2)
        return false;
    if(abs(marker2.pose.pose.position.z - marker1.pose.pose.position.z) > 2)
        return false;
```

Tras esto se miden las diferencias en orientación.

```
    tf::Quaternion q1(marker1.pose.pose.orientation.x,marker1.pose.pose.orientation.y,
                      marker1.pose.pose.orientation.z,marker1.pose.pose.orientation.w);
    tf::Matrix3x3 m1(q1);
    tf::Quaternion q2(marker2.pose.pose.orientation.x,marker2.pose.pose.orientation.y,
                      marker2.pose.pose.orientation.z,marker2.pose.pose.orientation.w);
    tf::Matrix3x3 m2(q2);

    m1.getRPY(roll1, pitch1, yaw1);
    m2.getRPY(roll2, pitch2, yaw2);

    if(abs(roll2 - roll1) > PI/2)
        return false;
    if(abs(pitch2 - pitch1) > PI/2)
        return false;
    if(abs(yaw2 - yaw1) > PI/2)
        return false;

    return true;
}
```

El problema de esto es que hay dos excepciones para las cuales este criterio no es válido. En primer lugar el marcador ha de ser el mismo. En segundo lugar, en caso de que se pierda el marcador de vista, en caso de que se volviese a mirar el marcador, el que estuviese almacenado no tiene por qué ser válido. Esto significa que además de comparar dos marcadores, se debe tener en cuenta el momento en el que se recibió el último marcador y además si el marcador es el mismo.

Todos estos factores construyen un pequeño problema lógico que se puede solucionar en primer lugar incluyendo una variable que contenga el momento en que se ha guardado el

último marcador. Lo siguiente es añadir un condicional que controle y compare todos estos factores y que controle si se publica y actualiza la información.

```
if (ar_pose_marker.id != _ARMarker.id || (ros::Time::now().toSec() - _time.toSec()) > 1
    || compareMarkers(_ARMarker, ar_pose_marker) == true)
```

En la figura 36, se muestra un flujograma del problema lógico.

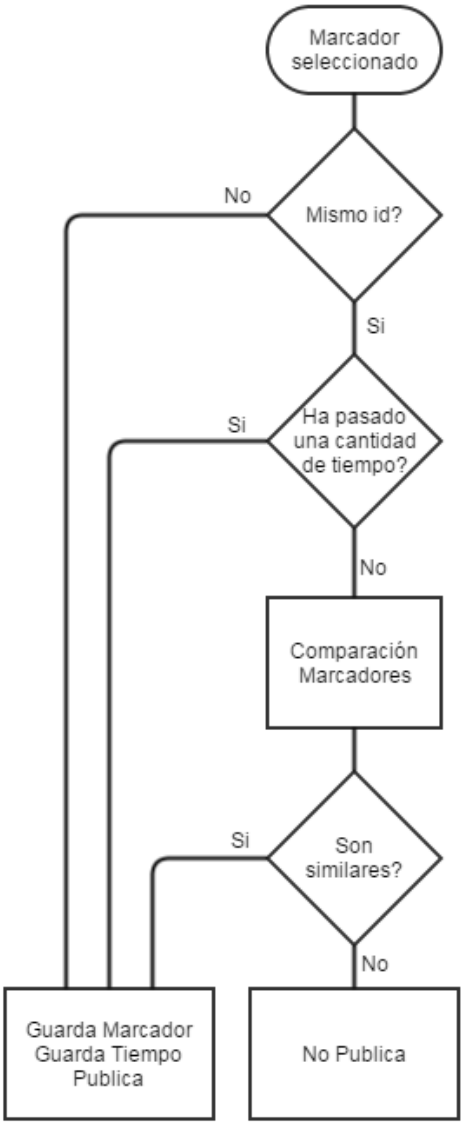


Figura 36. Flujograma Ampliación 2.

Esta lógica lo que representa es que se publicará siempre y cuando los marcadores sean distintos o haya pasado más de 1 segundo de tiempo desde la última actualización o los marcadores sean parecidos.

7. Pruebas

Con todo el sistema completo, el siguiente paso es comprobar la funcionalidad del sistema en distintas condiciones. En primer lugar se describirá el paquete auxiliar que se ha empleado para simular la odometría, que se subscribirá al nodo original. Tras esto, se mostrarán los resultados que se han obtenido con o sin las ampliaciones descritas en el capítulo anterior. Cada una de estas pruebas se desarrollará en un ambiente iluminado con luz natural y en otro iluminado con luz artificial, para observar la reacción del software en ambos casos.

7.1 Creación de una odometría falsa

En este punto del trabajo, se tiene completamente diseñado el sistema que se había propuesto en los objetivos iniciales. Este sistema tenía que calcular la posición global de una cámara para corregir el error de un sistema odométrico. Debido a que no se tiene la completa disponibilidad del vehículo en el que se encuentra implantado dicho sistema odométrico, se va a emplear una simulación prediseñada de dicha odometría. Esta simulación será un nodo de *ROS* llamado *odometry_publisher* que se subscribirá al *topic* publicado por el nodo *beacon_detection_node*. Este nodo publicará a su vez un sistema de coordenadas que se corresponderá con el de la cámara, que se irá moviendo en una dirección con una determinada velocidad, simulando el movimiento de la cámara, y en cuanto reciba información recibida a través del paquete *beacon_detection* en forma de transformada, tendrá que reubicar este sistema de coordenadas simulado. Una vez que deje de recibirlo, volverá a simular el movimiento pero esta vez desde la nueva ubicación y orientación recibida.

7.1.1 Diagrama de Flujo del Nodo *odometry_publisher*

En la figura 37 se puede ver un diagrama de flujo del nodo del paquete que simula la odometría falsa.

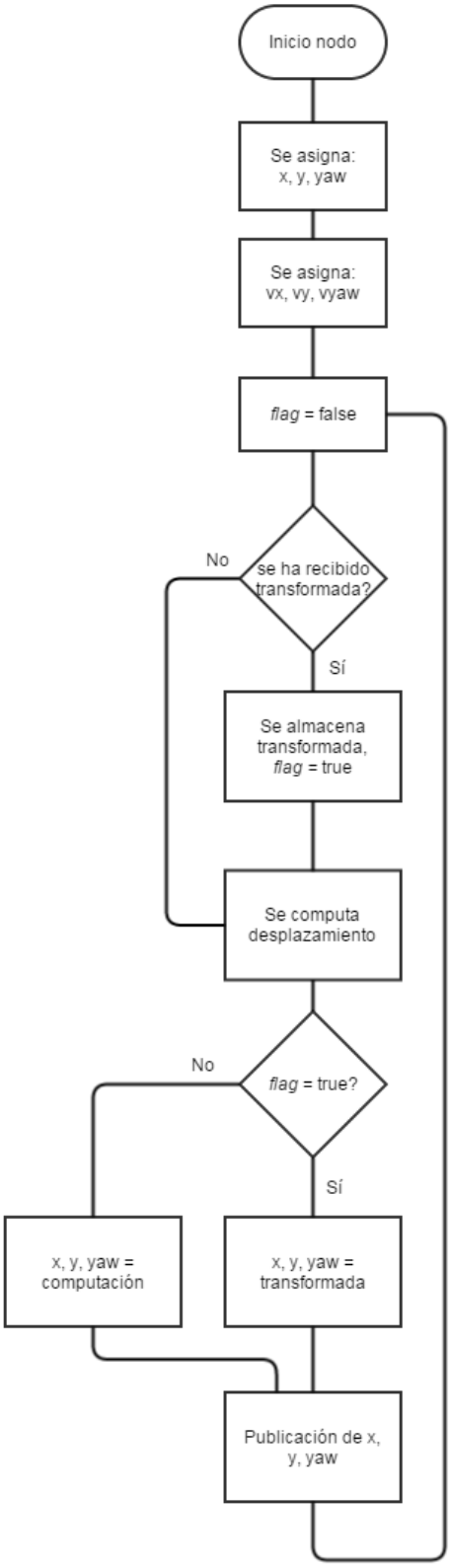


Figura 37. Flujograma nodo *odometry_publisher.*

Como se puede ver en el flujograma, la única información que se va a necesitar de la transformada que se envía a través del *topic WorldToCam,* desde el nodo *beacon_detection_node,* es la posición con respecto al eje "y" y al eje "x" y la orientación con respecto a *Yaw*. Esto se debe a que se asume que el centro del vehículo estará a ras de suelo y no se prevé desplazamiento en el vehículo en el eje "z" además de que no habrá más modificaciones en la orientación de las que haya en *Yaw*.

7.1.2 Subscripción a *beacon_detection*

El resultado que se espera obtener es la correcta subscripción del nodo que simula la odometría falsa al nodo creado *beacon_detection*. Y como puede ver en la figura 38, cuando se ejecuta el nodo *rqt_graph*, que ilustra gráficamente la Gráfica de Computación en *ROS*, el sistema al completo se encuentra correctamente conectado.

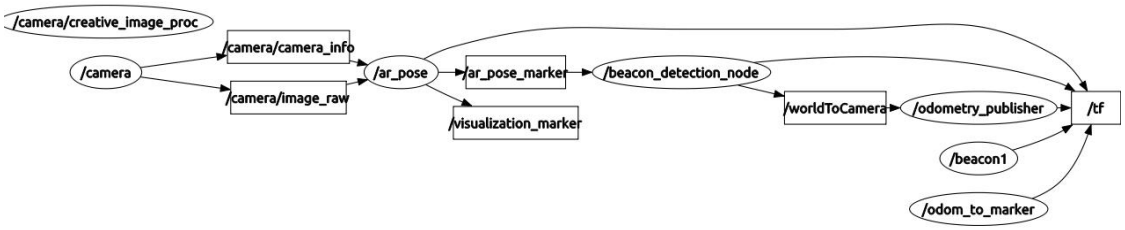


Figura 38. Sistema al completo con *odometry_publisher.*

7.2 Resultados

En este punto se ha incorporado la odometría falsa al sistema, por lo que se procede a realizar una serie de pruebas para comprobar que todo funciona correctamente, es decir, que el conjunto formado por la recepción de información, el procesamiento de los datos recibidos y la publicación de los mismos. Estas pruebas se van a localizar en dos entornos diferentes, marcados por dos características que como se ha visto afectan en gran manera a los métodos de visión por computador: la iluminación. Tras esto se ha procedido a hacer dos sesiones de pruebas, una sin ampliaciones, en la que se ha podido ver la necesidad de algunas mejoras, y la segunda sesión, que ha servido para corroborar que las ampliaciones estaban correctamente diseñadas.

7.2.1 Entornos de Pruebas

Se dispondrán de este modo dos entornos, uno con iluminación natural en el exterior y otro con iluminación artificial en interior. En cada uno de ellos se dispondrán un par de balizas y se realizará un pequeño recorrido para comprobar que todo funciona correctamente.

7.2.1.1 Entorno con luz natural

Este entorno controlado corresponde a un soportal con las siguientes medidas: 1.70 metros de ancho y 5 metro de largo. Se realizará un recorrido pasando por la primera y la segunda baliza. En la figura 39 se muestra el área completa.

Figura 39. Imágenes del entorno con luz natural.

Las balizas se encontrarán situadas con respecto al punto que se considera origen del siguiente modo:

- **Baliza 1:** $x = 1.5; y = -1.10; z = 1.20; Roll = 1.57; Pitch = 0; Yaw = 3.14;$

  En la figura 40 se muestra cómo se encuentra posicionada la primera baliza.

Figura 40. Posicionamiento de la primera baliza en entorno LN.

- **Baliza 2**: $x = 4; y = 0.5; z = 1.2; Roll = 1.57; Pitch = 0; Yaw = 0;$

  En la figura 41 se muestra donde está posicionada la segunda baliza.

Figura 41. Posicionamiento de la segunda baliza en entorno LN.

7.2.1.2 Entorno con luz artificial

En el segundo caso, el entorno se corresponde con un pasillo con las siguientes medidas: 0.90 m de ancho y 4.50 de largo. Se realizará un recorrido pasando por la primera y la segunda baliza del mismo modo que en el caso anterior. En la figura 42 se muestran imágenes del entorno al completo:

Figura 42. Imagen del entorno con luz artificial.

Las balizas se encontrarán situadas con respecto al punto que se considera origen del siguiente modo:

- **Baliza 1:** $x = 1.3; y = -0.4\,; z = 1.2; Roll = 1.57; Pitch = 0; Yaw = 3.14;$

  En la figura 43 se muestra cómo se encuentra posicionada la primera baliza.

Figura 43. Posicionamiento de la primera baliza en entorno LA.

- **Baliza 2:** $x = 4.2; y = 0.56; z = 1.2; Roll = 1.57; Pitch = 0; Yaw = 0;$

  En la figura 44 se muestra donde está posicionada la segunda baliza.

Figura 44. Posicionamiento de la segunda baliza en entorno LA.

7.2.2 Resultados iniciales

Una vez que ambos entornos se encuentran correctamente descritos ya se puede pasar a los resultados obtenidos en las primeras pruebas que se han realizado, previas a las ampliaciones. Las disposiciones de las balizas además de ser incluidas en el fichero *beacons.xml*, también se incorporan en el fichero *rviz_launcher.launch*, creando un par de balizas empleando el paquete *tf* que tendrán la misma información que en el *XML*, para que los resultados se puedan visualizar empleando *rviz*.

7.2.2.1 Resultados con luz natural

Como se ha indicado, la cámara se encontrará inicialmente en la posición inicial, coincidiendo con la posición que se ha denominado “odom”, que representa la referencia absoluta del sistema. Mientras que la cámara no detecte ningún marcador el nodo odometry_publisher publicará una transformada que representa a la cámara moviéndose a una determinada velocidad con una dirección cualquiera. La idea es acercarse a la primera baliza, avanzar ligeramente para que la cámara la pierda y vuelva a crear la ilusión de movimiento, pero desde esta última posición, y acabar visualizando la segunda baliza.

Como se puede ver en la figura 45, la cámara se encuentra en lo que se determina como posición inicial.

Figura 45. Posición Inicial de la cámara en el entorno con luz natural

Tras esto, la cámara se desplaza ligeramente hasta que comienza a encontrarse encuadrada la primera baliza en el marco, con un ángulo de al menos 80 grados, para que el programa sea capaz de distinguirla. En la figura 46 se muestra como la cámara visualiza la baliza y envía los resultados, que se muestran en el correspondiente entorno virtual que se ha creado con *rviz*.

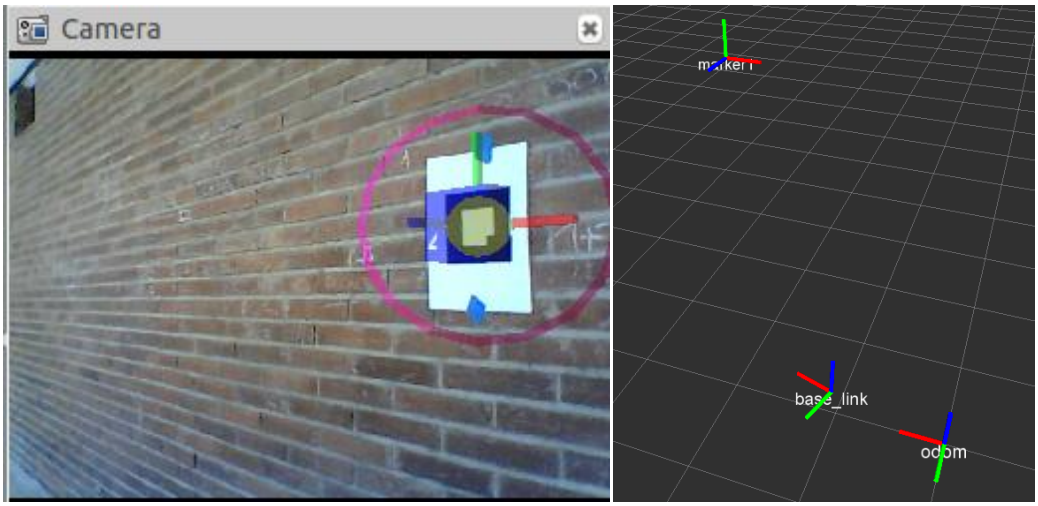


Figura 46. Reacción del sistema a la visualización de la primera baliza en entorno LN

Ahora, la cámara continúa su camino hacia la segunda baliza perdiendo de vista la primera. En la figura 47 se muestra cómo cuando se pierde de vista el marcador, se publica la transformada siguiendo la trayectoria que se había indicado en un primer momento pero desde el último punto en el que se vio la baliza.

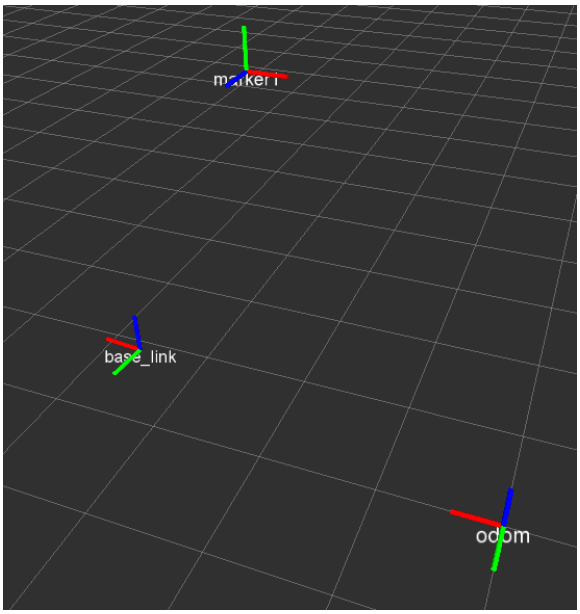


Figura 47. Pérdida de la visión de la primera baliza en entorno LN

En la figura 48, mostrada a continuación, se ha querido mostrar uno de los fenómenos que han motivado la creación de una de las ampliaciones. Durante las tomas de datos en algunos momentos han aparecido los denominados “outliers” que no dejan de ser falsos positivos que pueden surgir durante el recorrido. Esto ha ocurrido o bien mientras que la cámara estaba visualizando el objetivo o justo después de visualizarlo, durante un periodo menor a un segundo.

Figura 48. Aparición de “outliers” con luz natural

Por último, la cámara llega a la última baliza diferenciando cual de las dos es, debido a que el patrón que se encuentra dibujado en la misma es diferente, y de nuevo muestra la posición y orientación de la cámara en función de la referencia global “odom”. La figura 49 corrobora este hecho.

Figura 49. Reacción del sistema a la visualización de la segunda baliza en LN

A parte de verificar que todo ha funcionado correctamente se pueden sacar varias conclusiones de esta primera tanda de pruebas. En primer lugar y aunque se hayan realizado en un lugar cubierto, las condiciones de luz natural han aumentado la aparición de "outliers" y falsos positivos en general. Aunque los resultados han sido más que satisfactorios, este es un hecho que se ha de tener en cuenta. Aun así, las condiciones del entorno, en especial la opacidad del ladrillo donde se han situado las balizas, han permitido altos grados de detección. Como se comentó al describir *ar_pose*, la cualidad reflectante del lugar donde se sitúe el marcador es de gran relevancia en la capacidad de detección del software.

7.2.2.2 Resultados con luz artificial

Ahora se repite el mismo proceso pero en condiciones de interior, en el pasillo descrito anteriormente, un entorno aun más controlado que en el caso anterior. De nuevo el proceso es pasar entre balizas dando tiempo a la odometría a que se vuelva a "perder".

La posición inicial del recorrido se muestra en la figura 50.

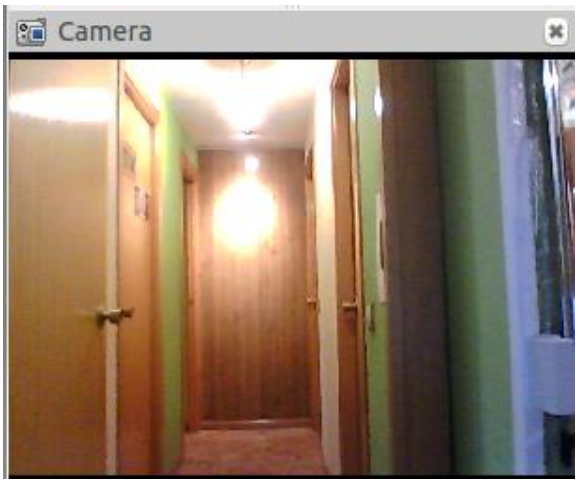


Figura 50. Posición inicial en el entorno con luz artificial

Tras esto se visualiza la primera baliza y el sistema de localización actúa en consecuencia relocalizando la posición de la cámara como se muestra en la figura 51.

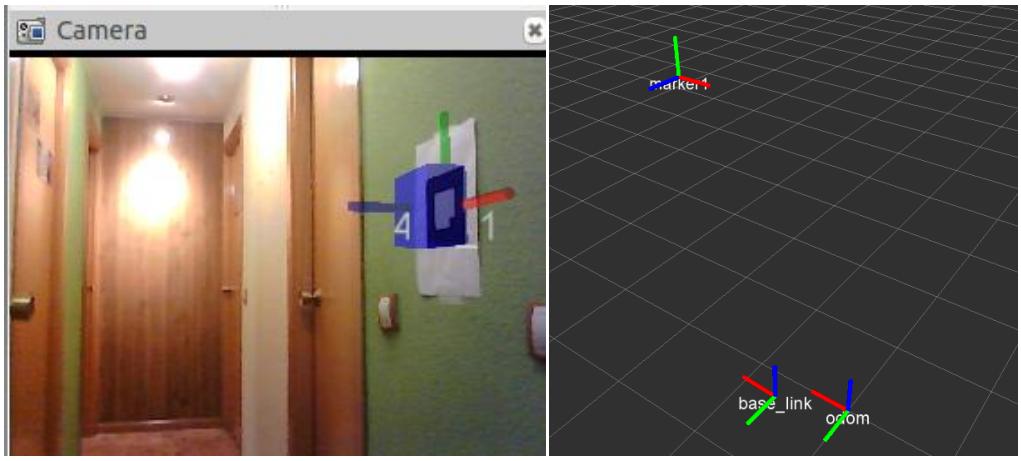


Figura 51. Detección de la primera baliza en el entorno LA

Se pierde ahora la visión de la primera baliza y el sistema comienza a simular de nuevo el movimiento de la cámara desde la última posición y orientación en la que se visualizó la baliza. La figura 52 ilustra este hecho.

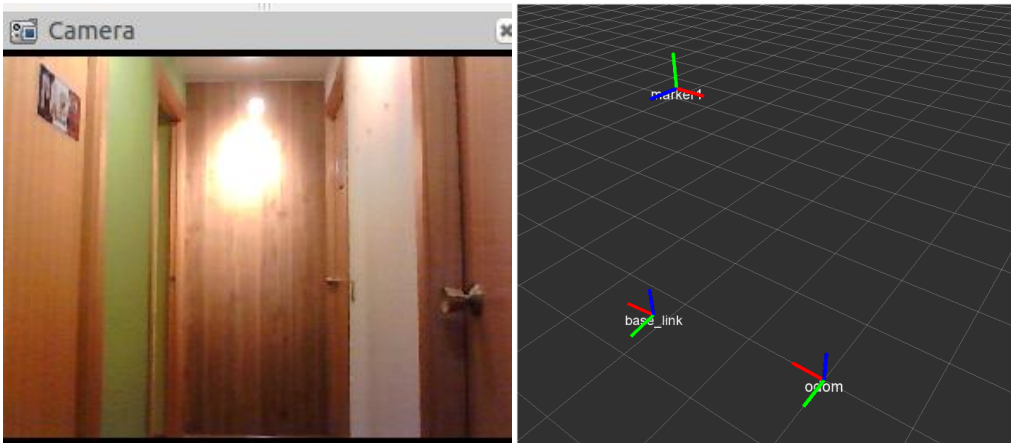


Figura 52. Pérdida de la primera baliza en el entorno LA

Por último se visualiza la última baliza. En la figura 53 se muestran los resultados.

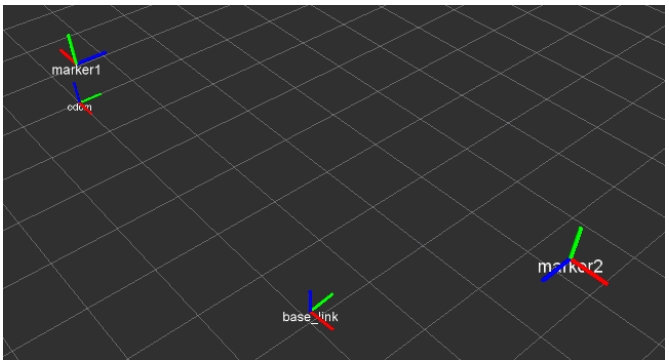


Figura 53. Detección de la segunda baliza en el entorno LA

En definitiva, los resultados son válidos y la detección es correcta. La elección del entorno vuelve a ser un factor importante ya que en este caso han aparecido menos “outliers”,

prácticamente ninguno, durante las pruebas. De nuevo se ha podido apreciar la diferencia de comportamiento del sistema a la hora de situar las balizas en según qué superficie.

7.2.3 Resultados con Ampliaciones

Como se ha visto a lo largo de la primera tanda de pruebas, la aparición de los denominados "outliers" durante la realización de las mismas ha motivado la realización de unos reajustes en el código. Aunque no afecta en absoluto a la detección como tal, si que cumple la función de filtro. De nuevo se repite el recorrido, pero esta vez no se va a prestar tanta atención a la cámara como tal sino al terminal. La ampliación que permite la filtración de "outliers" informa a través de un mensaje por terminal empleando lo que se denomina como *ROS: Warning*, que es un mensaje enfatizado, que se están filtrando los resultados.

7.2.3.1 Resultados con luz natural

Se realiza el recorrido, en primer lugar pasando por el marcador uno. En la figura 54 se muestra como al detectar el marcador en ocasiones filtra falsos positivos.

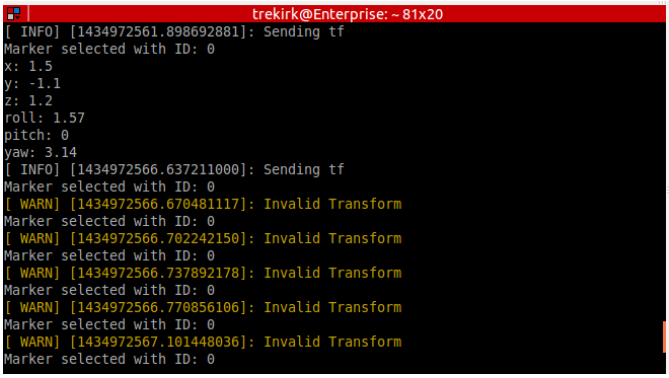


Figura 54. Filtración de outliers bajo luz natural

7.2.3.2 Resultados con luz artificial

Se ha repetido el proceso con el sistema con ampliaciones en el entorno con luz artificial. Y de nuevo se han filtrado resultados, pero en menor medida que en los casos con luz artificial, demostrando en definitiva que la lógica funciona correctamente. La figura 55 muestra algunos ejemplos de la mencionada filtración.

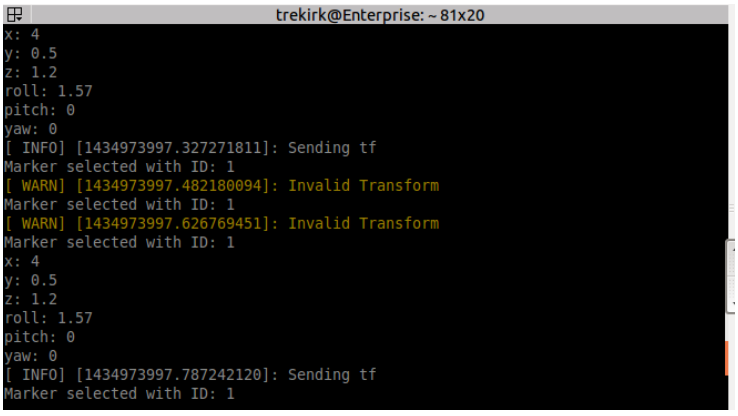


Figura 55. Filtración de outliers bajo luz artificial

8. Planificación y Presupuesto

En este capítulo se muestra la planificación que se ha seguido para desarrollar el proyecto en su totalidad. Para ello se expondrán las fases de desarrollo, se indicará el esfuerzo en cantidad de horas dedicadas por sección, y por último se mostrará la planificación detalladamente empleando un diagrama de Gantt.

8.1 Planificación

8.1.1 Fases de Desarrollo

El trabajo se encuentra dividido en 4 fases diferentes, en función de la tarea a realizar.

1. **Primera Fase: Formación.**

   En esta parte, el alumno recopila toda la información necesaria para realizar el proyecto. En primer lugar se trabaja con el sistema operativo en el que se va a trabajar inicialmente. Tras esto se trabaja en ROS, primero a nivel introductorio y después se realizan los tutoriales que permitan adquirir la soltura necesaria. Por último se investigan las técnicas de visión por computador, delimitando sus virtudes e inconvenientes y su adecuación al problema y el entorno planteado.

2. **Segunda Fase: Implementación.**

   Una vez que se ha trabajado teóricamente tanto en el entorno de trabajo como en el problema propuesto, se comienzan a intersecar estas dos facetas del proyecto. Se empiezan a implementar y probar las técnicas que se han investigado, y se comienza a conformar el sistema que se pretende diseñar, uniendo los primeros eslabones que lo forman.

3. **Tercera Fase: Diseño de la Aplicación.**

   En este punto la pieza que falta es la que tiene que diseñar el alumno, que se irá creando siguiendo unas pequeñas metas relacionadas con el funcionamiento de las aplicaciones en el entorno de programación ROS.

4. **Cuarta Fase: Pruebas.**

   Una vez que es sistema está completamente diseñado se hacen una serie de pruebas, se analizan los resultados y se sacan conclusiones. En función de estas pruebas se decide si hacer ampliaciones al código y, una vez que se amplia, se repite el proceso.

8.1.2 Horas dedicadas

Una vez que las fases de desarrollo están correctamente definidas, se ha de evaluar cuantas horas se han dedicado a cada fase, pudiendo dimensionar de este modo el esfuerzo del alumno y su distribución del mismo en función de la fase que se encontraba realizando.

A continuación se muestra el número de horas dedicadas por fase aproximadamente.

- **Fase de Formación:**

$$15\ semanas \times 11\ {}^{horas}/_{semana} = 165\ horas$$

- **Fase de Implementación:**

$$10\ semanas \times 13\ {}^{horas}/_{semana} = 130\ horas$$

- **Fase de Diseño de la Aplicación:**

$$17\ semanas \times 14\ {}^{horas}/_{semana} = 221\ horas$$

- **Fase de Pruebas**

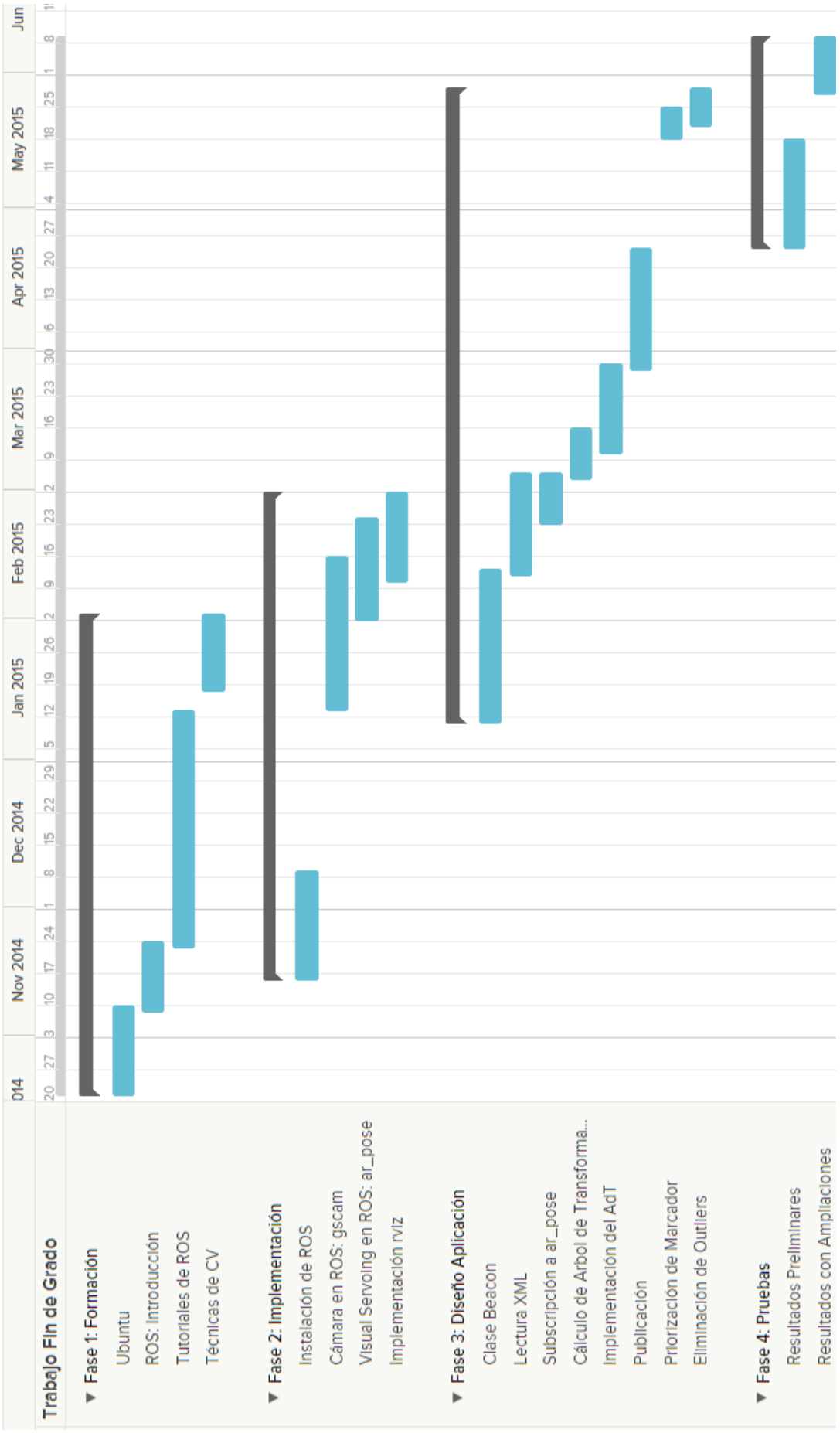
$$5\ semanas \times 10\ {}^{horas}/_{semana} = 50\ horas$$

El número de horas totales dedicadas al trabajo será la suma de las horas dedicadas por cada una de las fases.

$$H_{totales} = H_{f1} + H_{f2} + H_{f3} + H_{f4} = 566\ horas$$

Por último se muestra un diagrama de Gantt con la planificación correctamente desglosada.

8.1.3 Diagrama de Gantt



8.2 Presupuesto

8.2.1 Costes de Ejecución

En esta parte se desglosan todos los costes asociados al trabajo. Estando divididos en hardware, software y personal.

8.2.1.1 Costes por hardware

Hardware	Cantidad	Precio Ud. €	Amortización	Precio Total €
Portatil ASUS K53S	1	799.95	30%	239.98
				Total = 239.98

Tabla 3. Costes por Hardware

8.2.1.2 Costes por software

Software	Cantidad	Precio Ud. €	Precio Total €
Licencia *Ubuntu*	1	0	0
Licencia *ROS*	1	0	0
Licencia *TinyXml*	1	0	0
Licencia *ARToolKit*	1	0	0
			Total = 0

Tabla 4. Costes por Software

8.2.1.3 Costes por personal

Personal	Cantidad horas	Precio Ud. €	Precio Total €
Ingeniero junior	566	18	10188
			Total = 10188

Tabla 5. Costes por Personal

8.2.2 Importe Total

Costes	Precio Total €
Costes de Hardware	239.98
Costes de Software	0
Costes de Personal	10188
	Total = 10427.98

Tabla 6. Importe Total del Presupuesto

9. Conclusiones y Trabajos Futuros

Una vez que se ha realizado el proyecto al completo, este capítulo estará dedicado a compilar todas las conclusiones que se han sacado a lo largo del mismo, además se propondrán de una forma razonada ampliaciones asociadas a las limitaciones del sistema diseñado.

A lo largo del trabajo se han podido ver las posibilidades de la visión por computador. El *Visual Servoing* ha demostrado ser un método muy fiable y fácil de implementar, ya que a lo largo del proyecto no han surgido prácticamente complicaciones a la hora de incorporar este software, teniendo en cuenta que el hardware que se ha usado no se encuentra específicamente diseñado para este tipo de aplicaciones. Los resultados que se han ido obteniendo a lo largo de todo el proceso han demostrado esta facilidad de implementación ya que, aunque el proceso no ha sido simple como tal, no han surgido en ningún momento ningún tipo de inconvenientes. Las pruebas finales han demostrado como el sistema manda de manera adecuada información con altos grados de fiabilidad, exceptuando los "outliers", que aun así se han logrado eliminar en su mayoría.

En segundo lugar se ha de destacar el papel de *ROS* a lo largo de todo el trabajo, siendo un partícipe crucial durante el mismo. *ROS* no es simplemente un entorno de programación asociado a la robótica. Por un lado su sistema de programación modular permite esa versatilidad que actualmente se demanda en el campo de la robótica; la capacidad de poder crear y desarrollar sistemas cada vez más complejos incorporando software ya creado, evitando el tedioso proceso de "tener que reinventar la rueda". Este sistema además permite que crear sistemas sea algo mucho más intuitivo, dando prioridad a las piezas del rompecabezas, los nodos, y estandarizando las uniones entre estas piezas, los *topics*. La comunidad también juega un papel muy importante en *ROS*, permitiendo que el conocimiento se expanda, ya que, aunque se ha indicado antes que la versatilidad que permite el entorno es vital, de nada serviría si cualquier problema que surja durante la implementación no pudiera resolverse rápidamente. Todo esto combinado con el código abierto, permite el acceso a un campo tan pionero como es la robótica de un modo que era impensable hace no tanto tiempo. Este proyecto es una prueba de cómo a partir de algo tan cotidiano como es un portátil con una webcam, se puede contribuir en un proyecto mucho mayor y más complejo, como es un vehículo automatizado.

Se ha de destacar que se han cumplido todos los objetivos propuestos en el primer capítulo de esta memoria:

- Logrando implementar ROS, obteniendo todo el conocimiento requerido para ello.
- Desarrollando una aplicación inmersa en una red de comunicación de la que reciba información de su posición con respecto a una marca, además de recibir la información de la posición absoluta de dicha marca visual.
- Que dicha aplicación sea capaz de conjugar toda esta información, siendo capaz de finalmente computar su posición y orientación con respecto a un sistema de referencia absoluto.

- Enviando esta información con un cierto grado de fiabilidad, filtrando perturbaciones y así evitando en todo lo posible enviar información errónea.

El proyecto ha permitido adquirir y aplicar conocimientos ya adquiridos a lo largo de toda la carrera. Mientras que ya había una sólida formación en campos como la programación, el álgebra lineal y su aplicación industrial, se ha tenido la oportunidad de obtener conocimientos más específicos de los que no se tenía apenas noción, como puede ser en las técnicas de visión por computador y en la visión por computador en sí, o en entornos de programación y las comunicaciones en su interior, como se ha aprendido con *ROS*.

Por último, se ha de cerrar este capítulo hablando de los límites que se han tocado en este trabajo y qué ideas pueden ser llevadas a cabo para complementar el trabajo. En primer lugar, aunque el código tiene un método solvente para detectar incoherencias, solo puede hacerlo a partir de sus propios datos; si se quisiese un eliminación aún más alta de perturbaciones, se puede enriquecer el intercambio de información que se produce entre el software diseñado y el sistema de destino, de modo que el primero pueda trabajar con más referencias. Las condiciones técnicas con las que se ha trabajado, en especial la resolución de la cámara, influyen en la calidad de la detección, por tanto realizando mejoras en este apartado, se mejoraría el rendimiento de todo el sistema. En último lugar, se ha de indicar que el método que se ha empleado requiere que se modifique el entorno situando balizas para que el sistema logre orientarse. La combinación de este método con de percepción de objetos dotarían al sistema de una mayor adaptabilidad al entorno evitando el tener que modificarlo.

10. Referencias

[1] http://www.telegraph.co.uk/news/uknews/road-and-rail-transport/11403807/How-a-driverless-car-will-benefit-you.html

[2] World Healthcare Organization, "*Global Status Report on Road Safety*", 2013

[3] http://www.pcworld.es/movilidad/google-y-uber-avanzan-en-sus-coches-autonomos

[4] http://www.ros.org/about-ros/

[5] Linda G. Shapiro and George C. Stockman (2001). "*Computer Vision*". Prentice Hall. ISBN 0-13-030796-3

[6] Milan Sonka, Vaclav Hlavac and Roger Boyle (2008). "*Image Processing, Analysis, and Machine Vision*". Thomson. ISBN 0-495-08252-X.

[7] David A. Forsyth and Jean Ponce (2003). "*Computer Vision, A Modern Approach*". Prentice Hall. ISBN 0-13-085198-1.

[8] Aaron Martinez and Enrique Fernández (2013). "*Learning ROS for Robotics Programming",* Packt Publishing. ISBN 978-1-78216-144-8.

[9] F. Chaumette, S. Hutchinson. "*Visual Servo Control, Part I: Basic Approaches*". IEEE Robotics and Automation Magazine, 13(4):82-90, 2006

[10] Agin, G.J., *Real Time Control of a Robot with a Mobile Camera.* Technical Note 179, SRI International, 1979

[11] Danica Kragic and Henrik I Christensen, *Survey on Visual Servoing for Manipulation*

[12] Alexandre Krupa, Associate Member, IEEE, Jacques Gangloff, Member, IEEE, Christophe Doignon, Member, IEEE, Michel F. de Mathelin, Member, IEEE, Guillaume Morel, Member, IEEE, Joël Leroy, Luc Soler, and Jacques Marescaux, "*Autonomous 3-D Positioning of Surgical Instruments in Robotized Laparoscopic Surgery Using Visual Servoing*"

[13] *"ArDroneControl ROS package (SLAM and pose control for Ar.Drone)"*:

https://www.youtube.com/watch?v=c6fLsAwWbE0

[14] "*ARDrone Autonomous Landing Using Visual Servoing With Recovery Action":*

https://www.youtube.com/watch?v=iYPdP-9Axxc

[15] Ramesh Jain, Rangachar Kasturi, Brian G. Schunck, "*Machine Vision*", McGraw-Hill, Inc., ISBN 0-07-032018-7, 1995

[16] D.M. Gavrila, V. Philomin, "*Real-Time Object Detection for 'Smart' Vehicles"*

[17] Scaramuzza, D., Fraundorfer, F., "*Visual Odometry: Part I - The First 30 Years and Fundamentals*", IEEE Robotics and Automation Magazine, Volume 18, issue 4, 2011

[18] Maimone, M.; Cheng, Y.; Matthies, L. (2007). "*Two years of Visual Odometry on the Mars Exploration Rovers*" (PDF). *Journal of Field Robotics* **24** (3): 169–186.

[19] Johann Borenstein and Liqiang Feng, "*Measurement and Correction of Systematic Odometry Errors in Mobile Robots"*, IEEE Transactions on Robotics and Automation, Vol 12, No 6, December 1996, pp. 869-880.

[20] Campbell, J.; Sukthankar, R.; Nourbakhsh, I.; Pittsburgh, I.R."*Techniques for evaluating optical flow for visual odometry in extreme terrain*". Intelligent Robots and Systems, 2004.(I*ROS* 2004). Proceedings. 2004 IEEE/RSJ International Conference on **4**

[21] http://wiki.ros.org/viso2_ros?distro=hydro#I_run_mono_odometer_but_I_get_no_messages_on_the_output_topics

[22] https://www.asus.com/es/Notebooks_Ultrabooks/K53SV/specifications/

[23] http://wiki.ros.org/*ROS*/Concepts

[24] Stroustrup, Bjarne (1997). *"The C++ Programming Language"* (3ª edición). ISBN 0201889544.

[25] https://wiki.qt.io/Category:Tools::QtCreator_Spanish

[26] http://www.grinninglizard.com/tinyxmldocs/index.html

[27] http://wiki.ros.org/tf

[28] http://wiki.ros.org/gscam

[26] Kato, H., Billinghurst, M. "*Marker Tracking and HMD Calibration for a video-based Augmented Reality Conferencing System*". In Proceedings of the 2nd International Workshop on Augmented Reality (IWAR 99). October, San Francisco, USA. (1999)

[27] http://wiki.ros.org/visp_auto_tracker?distro=jade

[28] http://www.hitl.washington.edu/artoolkit/

[29] http://wiki.ros.org/ar_pose#Nodes (15/3/15)

[30] http://www.hitl.washington.edu/artoolkit/documentation/userarwork.htm

Anexos

Anexo I. Instalación de ROS

Una vez que los conceptos con los que se trabajará a lo largo del proyecto están debidamente explicados, se pasa a explicar cómo estará instalado *ROS* en el caso actual.

Debido a que es la versión más estable, se instalará *ROS* en su versión *Indigo* bajo el sistema operativo de *Linux*, *Ubuntu* en la versión 14.04. *ROS* pone a disposición del usuario unos completos tutoriales para instalar el entorno en casi cualquier sistema operativo, en este caso como es lógico se sigue el de Ubuntu. Para la instalación completa de ROS se introduce el siguiente comando bajo super-usuario:

```
sudo apt-get install ros-indigo-desktop-full
```

A la hora de trabajar con *ROS*, se ha de crear en primer lugar un espacio de trabajo o *workspace* donde se irán instalando los sucesivos paquetes. En este caso el nombre del *workspace* para que sea más intuitivo será *catkin_ws* y estará situado en la carpeta raíz. Dentro del *workspace*, una vez que ya se ha construido empleando la herramienta de *ROS catkin_make,* las carpetas estarán divididas en *build*, *devel* y *src*. Siendo de las tres carpetas la carpeta *src* (*source*) donde se instalarán los paquetes que se vayan a utilizar o a crear.

Dentro de la carpeta *src* como se aprecia en la siguiente imagen se encuentran distribuidos todos los paquetes de *ROS* que se quieran instalar en el *workspace*.

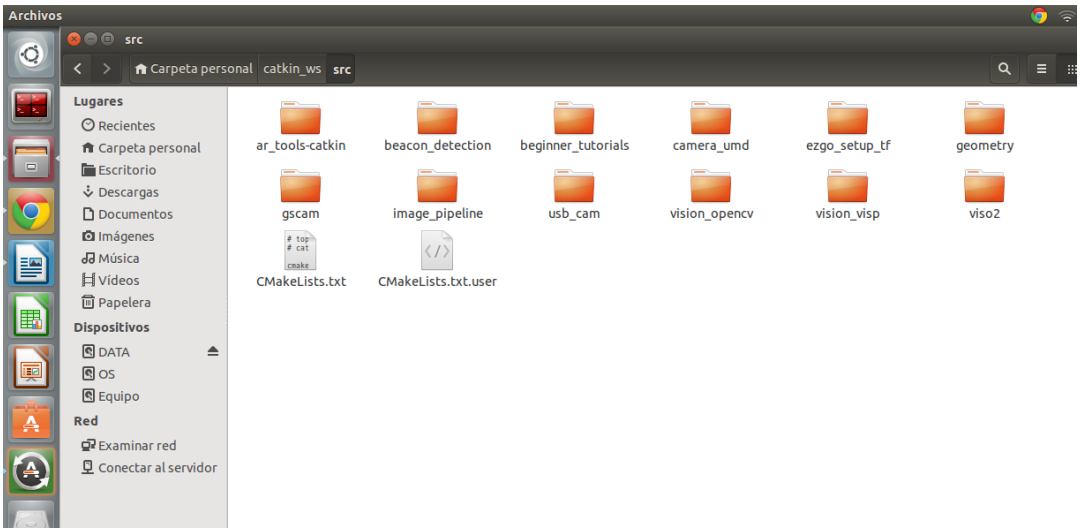


Figura 56. Carpeta *src* dentro de *catkin_ws.*

En la imagen siguiente se muestra la estructura interna del paquete que se va a crear para este proyecto, siendo similar a la estructura de la mayoría de los paquetes. Como se ha indicado, los paquetes pueden tener en su interior toda la información que se considere relevante para el susodicho paquete, pero siempre tendrán un *CMakeLists.txt* ya que en él se incluye la información que permite construir el paquete, y el manifiesto, que es el archivo que provee de toda la información necesaria acerca del paquete. Por lo general, el contenido del paquete

para que sea más intuitivo incluye una carpeta *source* o *src* donde se incluyen bibliotecas y ejecutables, y una carpeta *include*, donde se encuentran los *header*.

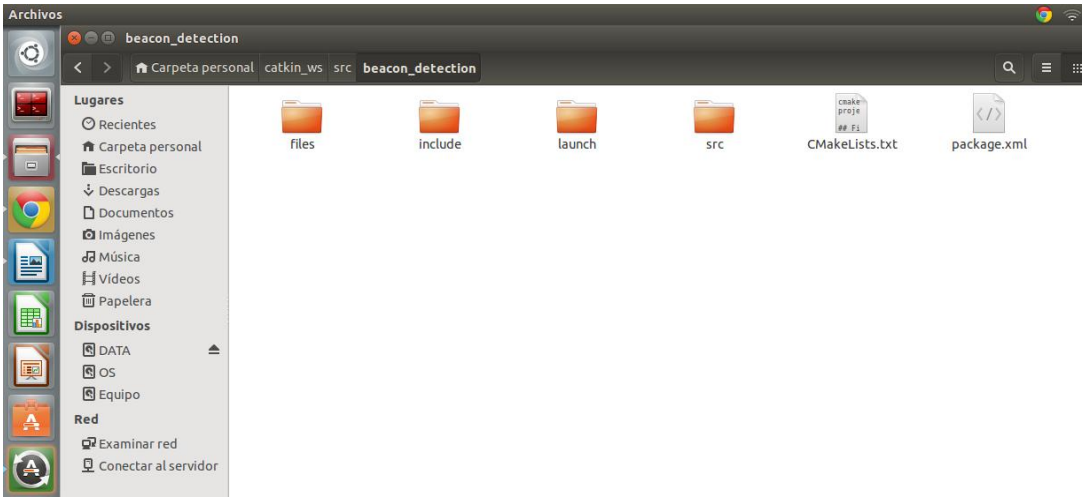


Figura 57. Paquete *de ROS.*

Anexo II. Calibración de la cámara

Una parte relevante a la hora de trabajar en *ROS* con drivers de cámaras es la calibración. *ROS* ofrece un paquete específicamente dedicado al trabajo de calibración y procesamiento de imagen compatible con prácticamente cualquier driver, en este caso con *gscam*. El nombre del paquete completo es *image_pipeline*. La calibración es un proceso bastante simple que lo único que requiere es la ejecución del paquete *camera_calibration,* dentro de *image_pipeline*, y la consecución de una serie de pasos.

En primer lugar lo que se requiere es la impresión de un tablero de ajedrez como el que se muestra en la imagen. El tamaño al que sea impreso es de libre elección, ya que es un parámetro que se indicará al ejecutar el nodo de calibración.

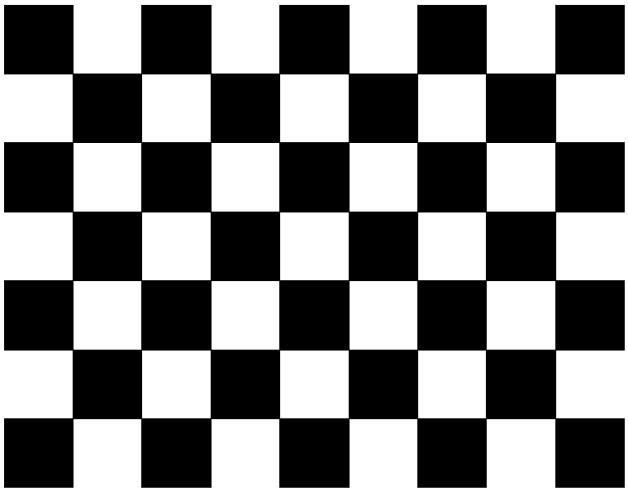


Figura 58. Patrón empleado en la calibración de la cámara.

Una vez que ya se tiene el tablero impreso y se saben sus medidas, se procede a ejecutar el driver de la cámara que se prefiera, para hacer que esta comience a publicar. Una vez que se verifiquen que los *topics* requeridos están siendo publicados, se introduce la siguiente línea de comando en el terminal.

```
rosrun camera_calibration cameracalibrator.py --size 8x6 --square
0.108 image:=/camera/image_raw camera:=/camera
```

Lo que representa esta línea de comando es la ejecución del nodo presente en *camera_calibration* con una serie de parámetros, que en este caso son en primer lugar el número de puntos entre cuadrados de ancho, por el número de puntos entre cuadrados de alto (en el caso del tablero que se ha empleado en este caso es de 6x8, como el del ejemplo), tras esto, se indica el ancho del cuadrado en metros, después se indica bajo que *topic* se publica la imagen, y por último el nombre del nodo que publica este *topic*.

A continuación se abrirá una ventana como la que se muestra en la siguiente imagen, el objetivo será ir moviendo el tablero de tal modo que la cámara sea capaz de localizar las intersecciones de los cuadrados en cualquier posición que se ponga dicho tablero. Una vez que

la información tomada sea suficiente, se guarda la información recogida, tanto en un fichero *.yaml* para poder cargarlo con cualquier driver o paquete, como en la propia cámara.

Figura 59. Respuesta del nodo *camera_calibration* a la aparición del patrón. [Fuente: http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration]

Para calibrar correctamente la cámara se ha de mover el patrón siguiendo un determinado orden alrededor del marco de la cámara:

- En la parte izquierda, derecha, superior e inferior del campo de visión de la cámara
    - Barra X – A la izquierda y derecha del campo de visión de la cámara
    - Barra Y – Parte superior e inferior del campo de visión de la cámara
    - Barra *Size* – cerca y lejos del campo de visión
- El patrón ha de estar cubriendo todo el campo de visión de la cámara.
- El patrón ha de estar ligeramente torcido hacia la dirección de la cámara, posicionado en las partes superior, inferior, izquierda y derecha de ésta

En cada paso se ha de verificar que el patrón es reconocido por la cámara. En las imágenes a continuación se muestra como se ha de posicionar el patrón a lo largo de la calibración.

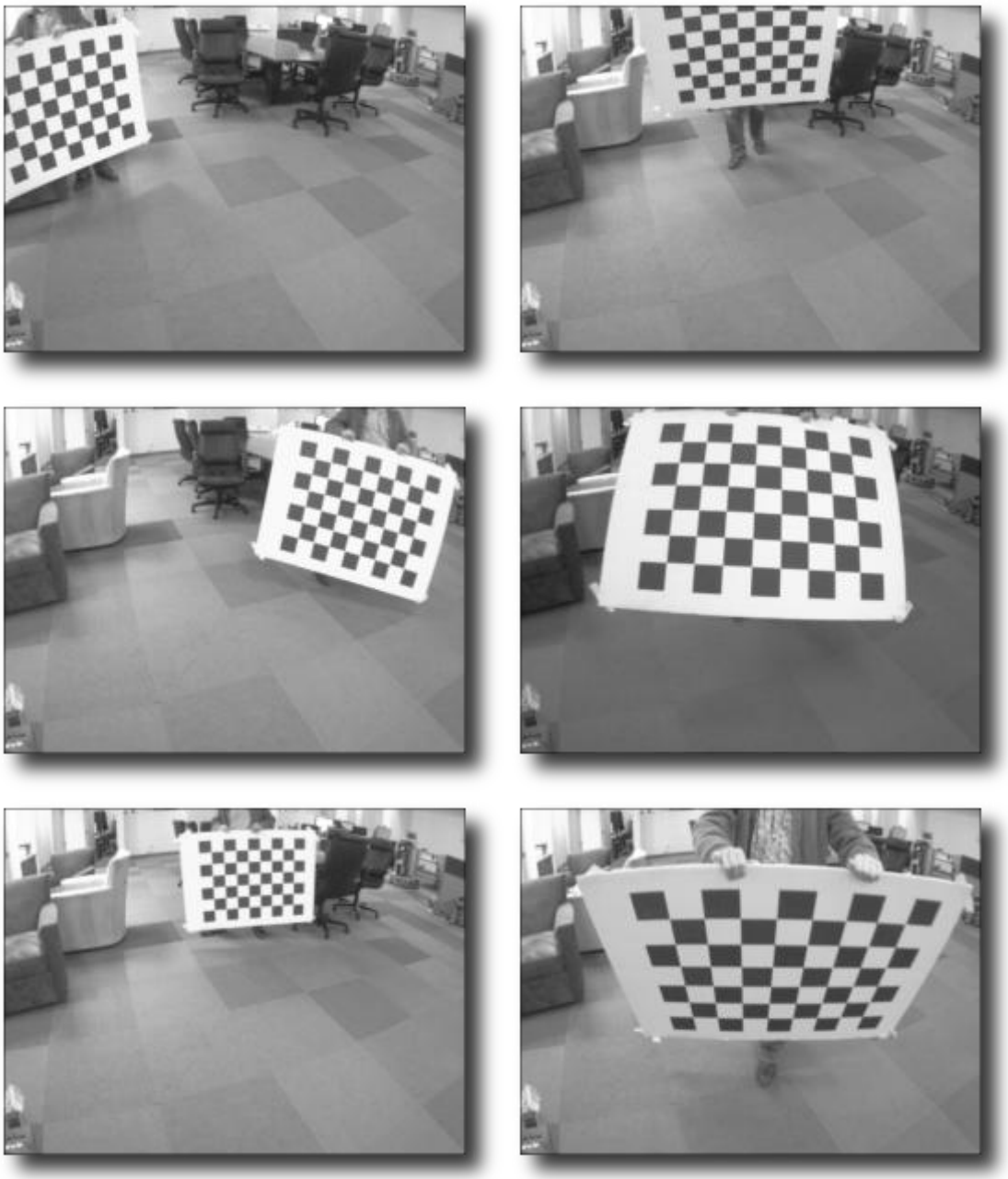


Figura 60. Distintos posicionamientos del patrón. [Fuente: http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration]

Una vez que se posean los datos de calibración necesarios, el botón *calibrate* se remarcará, indicando que la información puede ser almacenada.

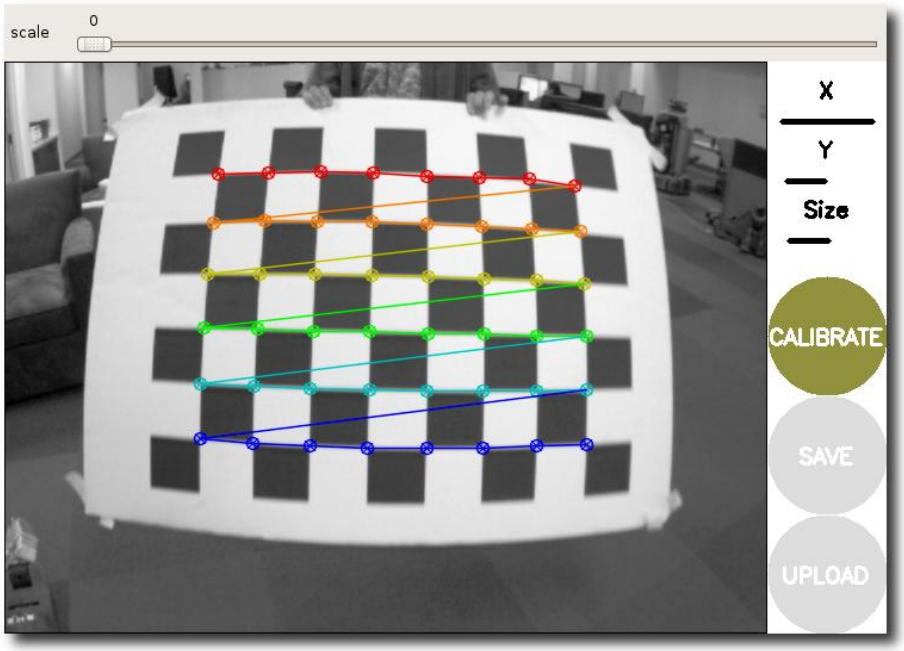


Figura 61. Botón *Calibrate* resaltado en el interfaz. [Fuente: http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration]

La información en este caso se guardara en el fichero *usb_cam.yaml* para emplearlo en caso de que un nodo lo necesite como parámetro.

Anexo III. package.xml de beacon_detection

Los archivos denominados como *package.xml*, son considerados en *ROS* manifiestos que dan toda la información relevante acerca del propio paquete. En este caso se hablará en particular del que se va a emplear para el trabajo, resaltando y justificando los campos más relevantes.

Lo primero que se ha de resaltar de un manifiesto en *ROS* son las dependencias. En *ROS* un paquete puede depender de varios paquetes, ya sea porque se subscribe a ellos o porque los emplea como bibliotecas. *ROS* posee la herramienta *rosdep* que permite, una vez se ha incorporado un paquete a un *workspace*, que *ROS* instale aquellos paquetes de los que dependa que se encuentren indicados en el manifiesto de dicho paquete. En el caso del paquete *beacon_detection*, éste dependerá de:

- ***roscpp*.** Debido a que es en el lenguaje en el que se va a programar
- ***std_msgs*.** Debido a que este programa se subscribirá.
- ***message_generation* y *message_runtime*.** Porque se pretende publicar con el paquete.
- ***ar_pose*.** Ya que es el paquete el que se va a subscribir, empleando además el objeto *ARMarker* en el código.
- ***tf*.** Que se empleará como biblioteca.

Para indicar una dependencia en el manifiesto, *ROS* separa la construcción y la ejecución, para indicar si el paquete poseerá relevancia dentro del código, o simplemente es relevante porque intercambia información con ese paquete cuando es ejecutado. La siguiente imagen muestra los dos tipos de dependencia en el caso de *ar_pose*:

```
<build_depend>ar_pose</build_depend>
<run_depend>ar_pose</run_depend>
```

El manifiesto contiene además otros campos de información personalizable con respecto al paquete, como por ejemplo el propio nombre del paquete, la versión del paquete, una descripción del mismo, el nombre y el e-mail del programador que lo mantiene o la licencia que posee el paquete.

Anexo IV. CMakeLists.txt de beacon_detection

El archivo *CMakeLists.txt* es la entrada hacia el sistema de construcción *CMake* para compilar y enlazar cualquier paquete de software. Cualquier paquete de *ROS* compatible con *CMake* contiene uno o más *CMakeLists.txt* en los que se describe cómo ha de ser construido el código y donde instalarlo. El archivo *CMakeLists.txt* empleado en los paquetes *catkin* es un *vanilla CmakeLists.txt* estándar con algunas restricciones adicionales.

Si para el package.xml la herramienta a destacar era *rosdep,* en el caso del *CMakeLists.txt* se destaca *catkin_make*. Se trata de una herramienta empleada en la línea de comandos que se emplea para construir código en un *workspace catkin*. Se ha de hacer la llamada a la herramienta desde la carpeta raíz del *workspace*.

Para que un *CMakeLists.txt* pueda ser correctamente construido dentro del sistema *catkin*, debe seguir un determinado formato. A continuación se muestran cada uno de los campos y su contenido en el caso de *beacon_detection*:

1. **Versión de CMake requerida** (cmake_minimum_required).

   cmake_minimum_required (VERSION 2.8.3)

2. **Nombre del Paquete** (project()).

   project(beacon_detection)

3. **Búsqueda de otros paqutes CMake/Catkin que se necesiten para la construcción del paquete**(find_package()).

   ```
   find_package(catkin REQUIRED COMPONENTS
       roscpp
       rospy
       std_msgs
       message_generation
       ar_pose
       tf
   )
   ```

4. **Generadores de Mensaje/Servicio/Accion** (add_message_files(), add_service_files(), add_action_files()).

5. **Invocación de generación de mensaje/servicio/accion** (generate_messages())

   ```
   generate_messages(
       DEPENDENCIES
       std_msgs
   )
   ```

6. **Especificar el tipo de construcción al que se va a someter** (catkin_package())

   ```
   catkin_package(
       CATKIN_DEPENDS message_runtime
   ```

```
)
```

7. **Localización e inclusión de directorios**(include_directories)

```
include_directories(include)
include_directories(
        ${catkin_INCLUDE_DIRS}
)
```

8. **Bibliotecas/Ejecutables a construir**
   (add_library()/add_executable()/target_link_libraries())

```
add_library(beacon_detection
        src/beacon.cpp
        src/tinyxml.cpp
        src/tinystr.cpp
        src/tinyxmlerror.cpp
        src/tinyxmlparser.cpp
)

add_executable(beacon_detection_node src/beacondetpublisher.cpp)
```

9. **Pruebas para buildear** (catkin_add_gtest())

10. **Reglas de instalación** (install())